DECSIM Reference Manual

This manual provides detailed descriptions of the syntax and functionality of the DECSIM Logic Simulator.

Revision/Update Information: This is a revised manual.
Software Version: V5.4
March 1989

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Copyright ©1989 by Digital Equipment Corporation

All Rights Reserved.
Printed in U.S.A.

The READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

- DEC
- DEC/CMS
- DEC/MMS
- DECnet
- DECsystem-10
- DECSYSTEM-20
- DECUS
- DECreter

- DIBOL
- EduSystem
- IAS
- MASSBUS
- PDP
- PDT
- RSTS
- RSX

- UNIBUS
- VAX
- VAXcluster
- VMS
- VT

This document was prepared using VAX DOCUMENT, Version 1.0.
PART 1  CONSTRUCTING AND CONNECTING MODELS

CHAPTER 1  HOW PRIMITIVES ARE DEFINED

1.1  EQUATION STATEMENTS
1.1.1 Environment
1.1.2 Definition
1.1.3 Components
  1.1.3.1 Keyword
  1.1.3.2 Parameters
  1.1.3.3 Output Name
  1.1.3.4 Equation Expression
1.1.4 Feedback in Equations
1.1.5 Restrictions
1.1.6 DECSIM Library Files

1.2  MOS PRIMITIVES
1.2.1 Environment
1.2.2 Definition
1.2.3 Components of the MOSFET Statement
  1.2.3.1 Keyword
  1.2.3.2 Parameters
  1.2.3.3 MOSFET Connections
1.2.4 Components of the RESISTOR Statement
  1.2.4.4 Keyword
  1.2.4.5 Parameters
  1.2.4.6 Connections
1.2.5 MOSFET and RESISTOR Restrictions
1.2.6 Length and Width Shrinkage

1.3  MEMORY PRIMITIVES
1.3.1 Environment
1.3.2 Definition
### 2.4 DECLARATIONS

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.1</td>
<td>PORT</td>
<td>2-12</td>
</tr>
<tr>
<td>2.4.1.1</td>
<td>Assigning Values to Ports</td>
<td>2-12</td>
</tr>
<tr>
<td>2.4.1.2</td>
<td>Reading from Output Ports</td>
<td>2-13</td>
</tr>
<tr>
<td>2.4.2</td>
<td>STATE</td>
<td>2-14</td>
</tr>
<tr>
<td>2.4.2.1</td>
<td>Communicating with States</td>
<td>2-15</td>
</tr>
<tr>
<td>2.4.2.2</td>
<td>Assigning Values to States</td>
<td>2-16</td>
</tr>
<tr>
<td>2.4.3</td>
<td>SYNONYM</td>
<td>2-16</td>
</tr>
<tr>
<td>2.4.4</td>
<td>CONSTANT</td>
<td>2-17</td>
</tr>
<tr>
<td>2.4.5</td>
<td>REVISION</td>
<td>2-18</td>
</tr>
<tr>
<td>2.4.6</td>
<td>REQUIRE</td>
<td>2-19</td>
</tr>
<tr>
<td>2.4.7</td>
<td>INFORM</td>
<td>2-19</td>
</tr>
<tr>
<td>2.4.8</td>
<td>MACRO</td>
<td>2-21</td>
</tr>
<tr>
<td>2.4.8.1</td>
<td>Defining Macros</td>
<td>2-19</td>
</tr>
<tr>
<td>2.4.8.2</td>
<td>Using Macros</td>
<td>2-21</td>
</tr>
<tr>
<td>2.4.9</td>
<td>ROUTINE</td>
<td>2-23</td>
</tr>
<tr>
<td>2.4.9.1</td>
<td>Routine Header</td>
<td>2-23</td>
</tr>
<tr>
<td>2.4.9.2</td>
<td>Return Value</td>
<td>2-24</td>
</tr>
<tr>
<td>2.4.9.3</td>
<td>Formal Parameters</td>
<td>2-24</td>
</tr>
<tr>
<td>2.4.9.4</td>
<td>Actual Parameters</td>
<td>2-25</td>
</tr>
<tr>
<td>2.4.9.5</td>
<td>Modifiers</td>
<td>2-26</td>
</tr>
<tr>
<td>2.4.9.6</td>
<td>Directives</td>
<td>2-26</td>
</tr>
<tr>
<td>2.4.10</td>
<td>FORWARD ROUTINE</td>
<td>2-26</td>
</tr>
<tr>
<td>2.4.11</td>
<td>INITIALIZE ROUTINE</td>
<td>2-26</td>
</tr>
<tr>
<td>2.4.12</td>
<td>EXTERNAL ROUTINE</td>
<td>2-27</td>
</tr>
<tr>
<td>2.4.13</td>
<td>INCLUDE</td>
<td>2-27</td>
</tr>
</tbody>
</table>

### 2.5 MODIFIERS

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.1</td>
<td>BIDIRECTIONAL</td>
<td>2-28</td>
</tr>
<tr>
<td>2.5.2</td>
<td>TWOSTATE</td>
<td>2-29</td>
</tr>
<tr>
<td>2.5.3</td>
<td>FOURSTATE</td>
<td>2-29</td>
</tr>
<tr>
<td>2.5.4</td>
<td>ENTRY</td>
<td>2-30</td>
</tr>
<tr>
<td>2.5.5</td>
<td>NOENTRY</td>
<td>2-30</td>
</tr>
<tr>
<td>2.5.6</td>
<td>CONVxx</td>
<td>2-31</td>
</tr>
<tr>
<td>2.5.7</td>
<td>UNCONNECTED_OK</td>
<td>2-32</td>
</tr>
<tr>
<td>2.5.8</td>
<td>STATIC</td>
<td>2-33</td>
</tr>
<tr>
<td>2.5.9</td>
<td>INITIAL(value)</td>
<td>2-34</td>
</tr>
<tr>
<td>2.5.10</td>
<td>INCLUDED</td>
<td>2-35</td>
</tr>
</tbody>
</table>

### 2.6 STATEMENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.1</td>
<td>BEGIN-END</td>
<td>2-36</td>
</tr>
<tr>
<td>2.6.2</td>
<td>IF-THEN, IF-THEN-ELSE</td>
<td>2-36</td>
</tr>
<tr>
<td>2.6.3</td>
<td>FOR</td>
<td>2-37</td>
</tr>
<tr>
<td>2.6.4</td>
<td>WHILE-DO, DO-WHILE</td>
<td>2-37</td>
</tr>
<tr>
<td>2.6.5</td>
<td>UNTIL-DO, DO-UNTIL</td>
<td>2-38</td>
</tr>
<tr>
<td>2.6.6</td>
<td>HALT</td>
<td>2-39</td>
</tr>
</tbody>
</table>

v
2.6.7 REPEAT .................................................. 2–41
2.6.8 SELECT ................................................ 2–42
2.6.9 SELECTONE .......................................... 2–43
2.6.10 SELECTALL ......................................... 2–44
2.6.11 RETURN ............................................. 2–45
2.6.12 LEAVE ................................................ 2–46
2.6.13 ACTIVATE ........................................... 2–47
2.6.14 %PRINT ............................................. 2–48
   2.6.14.1 WIDTH=integer • 2–49
   2.6.14.2 ZEROFILL • 2–49
   2.6.14.3 NOZEROFILL • 2–49
   2.6.14.4 RADIX=integer • 2–49
   2.6.14.5 LOGNAME=(logname_list) • 2–49
   2.6.14.6 NOCRLF • 2–49
   2.6.14.7 Print Elements • 2–49
2.6.15 WAIT ................................................ 2–51
2.6.16 Assignment ......................................... 2–52
2.6.17 Routine Call ....................................... 2–53
   2.6.17.1 Calling a Foreign Language Routine • 2–54
   2.6.17.2 Accessing DECSIM Data Structures from a Foreign
             Routine • 2–55
   2.6.17.3 Writing Foreign Language Routines • 2–56

2.7 EXPRESSIONS ........................................... 2–56
   2.7.1 Operands ........................................... 2–57
   2.7.2 Operators ......................................... 2–57
   2.7.3 Names ............................................. 2–57
   2.7.4 Literals ............................................. 2–57
   2.7.5 Unary Operations ................................ 2–58
   2.7.6 Binary Operations ................................. 2–58
   2.7.7 Routine Call ....................................... 2–58
   2.7.8 WAIT ................................................ 2–59
   2.7.9 Nested Expressions ................................. 2–59

2.8 DIRECTIVES ........................................... 2–59
   2.8.1 Scope of Directives ................................ 2–59
   2.8.2 TIMESCALE ......................................... 2–60
   2.8.3 MODULES ............................................ 2–60
   2.8.4 Operator Directives ............................... 2–60
   2.8.4.1 WIDTH • 2–60
   2.8.4.2 %CARRYIN • 2–61
   2.8.4.3 %CARRY • 2–61
   2.8.4.4 %OVERFLOW • 2–62
   2.8.4.5 Time • 2–62
   2.8.4.6 Arithmetic Representation • 2–63
### Contents

#### 2.9 TIMING DELAYS
- 2.9.1 Literals .................................................. 2–63
- 2.9.2 Expressions ............................................... 2–63
- 2.9.3 Formal Delay Parameters .............................. 2–64
- 2.9.4 Actual Delay Parameters .............................. 2–65

#### 2.10 SCHEDULING AND SEQUENCING STATEMENT EXECUTION
- 2.10.1 How the Scheduler Works .......................... 2–65
- 2.10.2 Timing and Sequencing Statement Execution .... 2–66

---

### CHAPTER 3 INTERCONNECT LANGUAGE

#### 3.1 NETWORK DESCRIPTION FILE
- 3.1.1 Library Format .......................................... 3–1
- 3.1.2 Network (Top Level) Format ......................... 3–2

#### 3.2 LIBRARY STATEMENT
- 3.2.1 LIBRARY Statement Defaults ....................... 3–2
- 3.2.2 Library Search Procedure ........................... 3–3

#### 3.3 MODEL DEFINITION
- 3.3.1 Model Definition Header ........................... 3–5
  - 3.3.1.1 Formal Parameters • 3–5
  - 3.3.1.2 Connection Lists • 3–6
  - 3.3.1.2.1 Signal–based Connection List • 3–6
  - 3.3.1.2.2 Pin–based Connection List • 3–7
- 3.3.2 Bidirectional Connections ........................... 3–7
- 3.3.3 GLOBAL Subblock ................................. 3–8
- 3.3.4 Model Body ............................................ 3–8
  - 3.3.4.1 Structure Subblock Components • 3–8
  - 3.3.4.2 Subblock Names • 3–9
- 3.3.5 Hierarchy in Model Definitions .................. 3–10

#### 3.4 INSTANCE STATEMENTS
- 3.4.1 Label ................................................... 3–10
- 3.4.2 Model Name ............................................ 3–10
- 3.4.3 Searching for the Model Definition to Instantiate . 3–11
- 3.4.4 Instance Actual Parameters ....................... 3–12
  - 3.4.4.1 Name–based Actual Parameters • 3–12
  - 3.4.4.2 Order–based Actual Parameters • 3–13
  - 3.4.4.3 Example of Actual Parameter Use • 3–13
## Contents

### 3.4.5 Instance Connection Lists
- 3.4.5.1 Name-based Instance Connection Lists • 3–14
- 3.4.5.2 Order-based Instance Connection Lists • 3–15
- 3.4.5.3 Implicit "label.pin" Signal Names • 3–15
- 3.4.5.4 Leaving Signals Unconnected • 3–16

### 3.4.6 Concatenating and Extracting Multibit Signals in Connection Lists
- 3.4.6.1 Concatenating Smaller Signals into a Larger Signal • 3–17
- 3.4.6.2 Extracting Smaller Signals from a Larger Signal • 3–19

### 3.5 INTERCONNECTING PRIMITIVES
- 3.5.1 Rules for Interconnecting Primitives • 3–21
- 3.5.2 Specifying the Type of Phantom • 3–21
- 3.5.3 How DECSIM Models Wire-tying
  - 3.5.3.1 Overview of the Modeling of Wire-tying • 3–22
  - 3.5.3.2 Insertion of Phantom Models into Your Network • 3–22
  - 3.5.3.3 User-defined Phantoms • 3–24

### 3.6 SIGNAL DECLARATIONS
- 3.6.1 Definition of Signal Declarations • 3–25
- 3.6.2 Model Header Signal Declarations • 3–26
  - 3.6.2.1 BIDIRECTIONAL Declaration • 3–26
  - 3.6.2.2 IN_STUB and OUT_STUB Declarations • 3–27
  - 3.6.2.3 UNCONNECTED_OK Declaration • 3–28
- 3.6.3 INPUT and OUTPUT Declarations • 3–29
- 3.6.4 SYNONYM Declaration • 3–30
- 3.6.5 Signal Declarations for Wire-tying
  - 3.6.5.1 WIREAND Signal Declaration • 3–31
  - 3.6.5.2 WIREOR Signal Declaration • 3–32
  - 3.6.5.3 TRISTATE, WIREMOS Signal Declarations • 3–33
  - 3.6.5.4 WIRE.MODELEVEL Signal Declaration • 3–34
- 3.6.6 TIMEOUT Declaration in Equation Primitives • 3–35
  - 3.6.6.1 NO_WIRETIE_CHECK Signal Declaration • 3–36
  - 3.6.6.2 PULLUP, PULLDOWN Signal Declarations • 3–36
  - 3.6.6.3 Consistency Checks for Wire-tying Declarations • 3–37
- 3.6.7 MOS Declarations • 3–37
  - 3.6.7.1 Specifying Node Attributes • 3–37
  - 3.6.7.2 POWER Declaration • 3–37
  - 3.6.7.3 CAPACITANCE Declaration • 3–38
  - 3.6.7.4 PARASITIC Declaration • 3–39
  - 3.6.7.5 PCAP Declaration • 3–40
  - 3.6.7.6 THRESHOLD Declaration • 3–40
  - 3.6.7.7 DELAY Declaration • 3–41
  - 3.6.7.8 NO_CHARGE_SHARE Declaration • 3–43
  - 3.6.7.9 TIMEOUT Declaration in MOS Primitives • 3–43
  - 3.6.7.10 EXPLICIT_NODE Declaration • 3–44
## PART 2  COMPILING AND SIMULATING MODELS

### CHAPTER 4  DECSIM ENVIRONMENT AND CONVENTIONS  4-1

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>COMMAND LANGUAGE CONVENTIONS</td>
<td>4-2</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Command Format</td>
<td>4-2</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Abbreviations</td>
<td>4-3</td>
</tr>
<tr>
<td>4.1.2.1</td>
<td>Command Keyword Abbreviations</td>
<td>4-3</td>
</tr>
<tr>
<td>4.1.2.2</td>
<td>Symbol Name Abbreviations</td>
<td>4-4</td>
</tr>
<tr>
<td>4.1.3</td>
<td>End-of-Line Comment (!)</td>
<td>4-4</td>
</tr>
<tr>
<td>4.1.4</td>
<td>End-of-Line Continuations</td>
<td>4-5</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Labels (Command Labels)</td>
<td>4-5</td>
</tr>
<tr>
<td>4.1.6</td>
<td>Errors</td>
<td>4-6</td>
</tr>
<tr>
<td>4.1.6.1</td>
<td>Error Handling of Syntax Errors</td>
<td>4-6</td>
</tr>
<tr>
<td>4.1.6.2</td>
<td>Semantic Error Handling</td>
<td>4-7</td>
</tr>
<tr>
<td>4.1.6.3</td>
<td>Error Handling for Expressions, EVALUATE</td>
<td>4-7</td>
</tr>
<tr>
<td>4.1.6.4</td>
<td>Error Reporting During Indirect File Execution</td>
<td>4-8</td>
</tr>
<tr>
<td>4.1.6.5</td>
<td>Error Reporting During Execution of Blocks and Loops</td>
<td>4-8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>COMMAND ARGUMENTS</td>
<td>4-8</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Quoted Text Strings</td>
<td>4-9</td>
</tr>
<tr>
<td>4.2.2</td>
<td>File Specifications and DECSIM Log Names</td>
<td>4-10</td>
</tr>
<tr>
<td>4.2.2.1</td>
<td>Special Characters in File Names</td>
<td>4-10</td>
</tr>
<tr>
<td>4.2.2.2</td>
<td>Default Directory and Paths</td>
<td>4-10</td>
</tr>
<tr>
<td>4.2.2.3</td>
<td>Default File Extensions</td>
<td>4-10</td>
</tr>
<tr>
<td>4.2.2.4</td>
<td>DECSIM Log Names</td>
<td>4-10</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Logic Values</td>
<td>4-11</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Integer Numeric Values</td>
<td>4-13</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Time Values</td>
<td>4-13</td>
</tr>
<tr>
<td>4.2.6</td>
<td>Expression Components</td>
<td>4-14</td>
</tr>
<tr>
<td>4.2.6.1</td>
<td>Expression Operands</td>
<td>4-14</td>
</tr>
<tr>
<td>4.2.6.2</td>
<td>Expression Constants</td>
<td>4-14</td>
</tr>
<tr>
<td>4.2.6.3</td>
<td>System Variables</td>
<td>4-15</td>
</tr>
<tr>
<td>4.2.6.4</td>
<td>State Variables Defined by the STATE Command</td>
<td>4-20</td>
</tr>
<tr>
<td>4.2.6.5</td>
<td>Subexpressions</td>
<td>4-20</td>
</tr>
<tr>
<td>4.2.6.6</td>
<td>Expression Operators</td>
<td>4-20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>ACCESSING SYMBOLS IN DECSIM</td>
<td>4-20</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Access to Unnamed Signals</td>
<td>4-21</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Access Lists in DECSIM Commands</td>
<td>4-21</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Hierarchical Access to Symbols</td>
<td>4-22</td>
</tr>
<tr>
<td>4.3.4</td>
<td>SET SCOPE Command</td>
<td>4-23</td>
</tr>
<tr>
<td>4.3.5</td>
<td>The &quot;%NET&quot; Pseudo-label</td>
<td>4-23</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.3.6</td>
<td>Accessing Symbols Using Wildcard Characters</td>
<td>4-23</td>
</tr>
<tr>
<td>4.3.7</td>
<td>Specific Symbol Types</td>
<td>4-25</td>
</tr>
<tr>
<td>4.3.7.1</td>
<td>Instance Labels • 4-25</td>
<td></td>
</tr>
<tr>
<td>4.3.7.2</td>
<td>Signal Names • 4-25</td>
<td></td>
</tr>
<tr>
<td>4.3.7.3</td>
<td>Specifying Assertion and Polarity • 4-26</td>
<td></td>
</tr>
<tr>
<td>4.3.7.4</td>
<td>Signal Names in a Hierarchy • 4-27</td>
<td></td>
</tr>
<tr>
<td>4.3.7.5</td>
<td>Behavior Global States • 4-28</td>
<td></td>
</tr>
<tr>
<td>4.3.7.6</td>
<td>Behavior Local States • 4-28</td>
<td></td>
</tr>
<tr>
<td>4.3.7.7</td>
<td>Behavior Routine Names • 4-29</td>
<td></td>
</tr>
<tr>
<td>4.3.7.8</td>
<td>Behavior Routine Formal Parameter Names • 4-29</td>
<td></td>
</tr>
<tr>
<td>4.3.7.9</td>
<td>Memory Array Names • 4-29</td>
<td></td>
</tr>
<tr>
<td>4.3.7.10</td>
<td>Command States • 4-29</td>
<td></td>
</tr>
<tr>
<td>4.3.8</td>
<td>Machine Descriptors</td>
<td>4-30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4.4</th>
<th>INTERACTIVE FEATURES</th>
<th>4-32</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.1</td>
<td>Command Modes and Command Prompting</td>
<td>4-32</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Escape Prompting</td>
<td>4-33</td>
</tr>
<tr>
<td>4.4.3</td>
<td>CTRL/G</td>
<td>4-34</td>
</tr>
<tr>
<td>4.4.4</td>
<td>CTRL/C Emergency Interrupt</td>
<td>4-34</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Break Points</td>
<td>4-36</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4.5</th>
<th>SPECIALIZED USES OF DECSIM</th>
<th>4-36</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5.1</td>
<td>Fault Simulation</td>
<td>4-36</td>
</tr>
<tr>
<td>4.5.2</td>
<td>DECSIM and ZYCAD</td>
<td>4-37</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Hazard Analysis</td>
<td>4-39</td>
</tr>
<tr>
<td>4.5.3.1</td>
<td>Features • 4-40</td>
<td></td>
</tr>
<tr>
<td>4.5.3.2</td>
<td>Restrictions • 4-40</td>
<td></td>
</tr>
<tr>
<td>4.5.3.3</td>
<td>Hazard Analysis in the Design Process • 4-40</td>
<td></td>
</tr>
<tr>
<td>4.5.3.4</td>
<td>Hazard Checker Violations and Boundary Conditions • 4-42</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
<th>COMMAND DESCRIPTIONS</th>
<th>5-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVATE</td>
<td></td>
<td>5-2</td>
</tr>
<tr>
<td>BEGIN-END</td>
<td></td>
<td>5-5</td>
</tr>
<tr>
<td>BREAK</td>
<td></td>
<td>5-7</td>
</tr>
<tr>
<td>CANCEL BREAK</td>
<td></td>
<td>5-9</td>
</tr>
<tr>
<td>DISABLE/ENABLE BREAK</td>
<td></td>
<td>5-11</td>
</tr>
<tr>
<td>SHOW BREAK</td>
<td></td>
<td>5-12</td>
</tr>
<tr>
<td>CALL</td>
<td></td>
<td>5-13</td>
</tr>
<tr>
<td>COMBEH (COMMAND MACRO)</td>
<td></td>
<td>5-15</td>
</tr>
<tr>
<td>COMPILE</td>
<td></td>
<td>5-18</td>
</tr>
<tr>
<td>CONFIGURE</td>
<td></td>
<td>5-23</td>
</tr>
<tr>
<td>CONTEXT</td>
<td></td>
<td>5-25</td>
</tr>
<tr>
<td>CONTINUE (COMMAND MACRO)</td>
<td></td>
<td>5-27</td>
</tr>
<tr>
<td>DCL</td>
<td></td>
<td>5-29</td>
</tr>
</tbody>
</table>
## Contents

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG</td>
<td>5–30</td>
</tr>
<tr>
<td>DEPOSIT</td>
<td>5–32</td>
</tr>
<tr>
<td>DETECT</td>
<td>5–36</td>
</tr>
<tr>
<td>CANCEL DETECT</td>
<td>5–50</td>
</tr>
<tr>
<td>DISABLE/ENABLE DETECT</td>
<td>5–51</td>
</tr>
<tr>
<td>SHOW DETECT</td>
<td>5–52</td>
</tr>
<tr>
<td>DROP</td>
<td>5–56</td>
</tr>
<tr>
<td>EDIT</td>
<td>5–57</td>
</tr>
<tr>
<td>EDITCOMMANDMACRO (COMMAND MACRO)</td>
<td>5–59</td>
</tr>
<tr>
<td>EDITTEXTMACRO (COMMAND MACRO)</td>
<td>5–60</td>
</tr>
<tr>
<td>ERROR</td>
<td>5–61</td>
</tr>
<tr>
<td>EVALUATE (EXPRESSION EVALUATION)</td>
<td>5–64</td>
</tr>
<tr>
<td>EXAMINE</td>
<td>5–66</td>
</tr>
<tr>
<td>EXIT AND QUIT</td>
<td>5–73</td>
</tr>
<tr>
<td>FAULT</td>
<td>5–75</td>
</tr>
<tr>
<td>CANCEL FAULT</td>
<td>5–82</td>
</tr>
<tr>
<td>SHOW FAULT</td>
<td>5–84</td>
</tr>
<tr>
<td>FOR</td>
<td>5–88</td>
</tr>
<tr>
<td>HALT</td>
<td>5–90</td>
</tr>
<tr>
<td>HAZARD</td>
<td>5–91</td>
</tr>
<tr>
<td>CANCEL, DISABLE, ENABLE HAZARD</td>
<td>5–102</td>
</tr>
<tr>
<td>SHOW HAZARD</td>
<td>5–105</td>
</tr>
<tr>
<td>HELP</td>
<td>5–108</td>
</tr>
<tr>
<td>IF</td>
<td>5–110</td>
</tr>
<tr>
<td>INDIRECT (@)</td>
<td>5–112</td>
</tr>
<tr>
<td>CANCEL INDIRECT</td>
<td>5–114</td>
</tr>
<tr>
<td>DISABLE/ENABLE INDIRECT</td>
<td>5–115</td>
</tr>
<tr>
<td>SHOW INDIRECT</td>
<td>5–117</td>
</tr>
<tr>
<td>INFORM (COMMAND MACRO)</td>
<td>5–118</td>
</tr>
<tr>
<td>KEY</td>
<td>5–119</td>
</tr>
<tr>
<td>SHOW KEY</td>
<td>5–122</td>
</tr>
<tr>
<td>LABELS (COMMAND MACRO)</td>
<td>5–124</td>
</tr>
<tr>
<td>LEAVE</td>
<td>5–125</td>
</tr>
<tr>
<td>LOAD</td>
<td>5–127</td>
</tr>
<tr>
<td>LOG</td>
<td>5–134</td>
</tr>
<tr>
<td>CANCEL LOG</td>
<td>5–139</td>
</tr>
<tr>
<td>SHOW LOG</td>
<td>5–141</td>
</tr>
<tr>
<td>DISABLE/ENABLE LOG</td>
<td>5–142</td>
</tr>
<tr>
<td>MACRO</td>
<td>5–143</td>
</tr>
<tr>
<td>SHOW MACRO</td>
<td>5–148</td>
</tr>
<tr>
<td>CANCEL MACRO</td>
<td>5–150</td>
</tr>
<tr>
<td>MOS</td>
<td>5–151</td>
</tr>
<tr>
<td>Term</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------</td>
</tr>
<tr>
<td>PATTERN</td>
<td>5–153</td>
</tr>
<tr>
<td>PAUSE (COMMAND MACRO)</td>
<td>5–163</td>
</tr>
<tr>
<td>PRINT</td>
<td>5–164</td>
</tr>
<tr>
<td>RADIX</td>
<td>5–169</td>
</tr>
<tr>
<td>RECALL</td>
<td>5–171</td>
</tr>
<tr>
<td>REPLAY</td>
<td>5–173</td>
</tr>
<tr>
<td>RESTORE</td>
<td>5–175</td>
</tr>
<tr>
<td>RESUME (COMMAND MACRO)</td>
<td>5–176</td>
</tr>
<tr>
<td>SAVE</td>
<td>5–177</td>
</tr>
<tr>
<td>SCOPE</td>
<td>5–179</td>
</tr>
<tr>
<td>SELECT</td>
<td>5–182</td>
</tr>
<tr>
<td>SHARE</td>
<td>5–184</td>
</tr>
<tr>
<td>SIMULATE</td>
<td>5–192</td>
</tr>
<tr>
<td>SPAWN</td>
<td>5–196</td>
</tr>
<tr>
<td>SPR (COMMAND MACRO)</td>
<td>5–199</td>
</tr>
<tr>
<td>STATE</td>
<td>5–200</td>
</tr>
<tr>
<td>STATISTICS</td>
<td>5–202</td>
</tr>
<tr>
<td>STEP</td>
<td>5–205</td>
</tr>
<tr>
<td>SYMBOLS (COMMAND MACRO)</td>
<td>5–207</td>
</tr>
<tr>
<td>TERMINAL</td>
<td>5–208</td>
</tr>
<tr>
<td>TIME</td>
<td>5–210</td>
</tr>
<tr>
<td>TIMING</td>
<td>5–212</td>
</tr>
<tr>
<td>TRACE</td>
<td>5–214</td>
</tr>
<tr>
<td>CANCEL/DISABLE/ENABLE TRACE</td>
<td>5–226</td>
</tr>
<tr>
<td>SHOW TRACE</td>
<td>5–228</td>
</tr>
<tr>
<td>TYPE (COMMAND MACRO)</td>
<td>5–229</td>
</tr>
<tr>
<td>UNTIL</td>
<td>5–230</td>
</tr>
<tr>
<td>USE</td>
<td>5–231</td>
</tr>
<tr>
<td>VAXSTATISTICS (COMMAND MACRO)</td>
<td>5–233</td>
</tr>
<tr>
<td>VMS_COMMAND (COMMAND MACRO)</td>
<td>5–236</td>
</tr>
<tr>
<td>WATCH</td>
<td>5–238</td>
</tr>
<tr>
<td>CANCEL/DISABLE/ENABLE WATCH</td>
<td>5–244</td>
</tr>
<tr>
<td>SHOW WATCH</td>
<td>5–246</td>
</tr>
<tr>
<td>WHILE AND UNTIL</td>
<td>5–248</td>
</tr>
<tr>
<td>ZYCAD</td>
<td>5–250</td>
</tr>
</tbody>
</table>
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.10.3</td>
<td>Binary Boolean Operators (Multiple-Bit Result)</td>
</tr>
<tr>
<td>D.11</td>
<td>CONCATENATION OPERATOR</td>
</tr>
<tr>
<td>D.12</td>
<td>CONVERSION OPERATORS</td>
</tr>
<tr>
<td>D.12.1</td>
<td>Binary Conversions</td>
</tr>
<tr>
<td>D.12.2</td>
<td>High-Impedance and Undefined Conversions</td>
</tr>
<tr>
<td>D.13</td>
<td>EXTENSION OPERATORS</td>
</tr>
<tr>
<td>D.14</td>
<td>MAPPING OPERATOR</td>
</tr>
<tr>
<td>D.15</td>
<td>PARENTHESSES OPERATOR</td>
</tr>
<tr>
<td>D.16</td>
<td>RELATIONAL OPERATORS</td>
</tr>
<tr>
<td>D.17</td>
<td>SHIFT OPERATORS</td>
</tr>
<tr>
<td>D.18</td>
<td>TRISTATE OPERATORS</td>
</tr>
</tbody>
</table>

**APPENDIX E**  
**ZYCAD PARTITIONS**  

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.1</td>
<td>ZYCAD LOGIC EVALUATOR ARCHITECTURE</td>
</tr>
<tr>
<td>E.1.1</td>
<td>Bottlenecks</td>
</tr>
<tr>
<td>E.1.2</td>
<td>How the DECSIM/ZYCAD Interface Partitions a Network</td>
</tr>
<tr>
<td>E.2</td>
<td>HOW TO PARTITION</td>
</tr>
<tr>
<td>E.2.1</td>
<td>Syntax of Partition Input (PAI) File</td>
</tr>
<tr>
<td>E.2.2</td>
<td>Sample Partition Output Files</td>
</tr>
<tr>
<td>E.2.3</td>
<td>Where Partitioning Can Fail</td>
</tr>
<tr>
<td>E.2.4</td>
<td>Tuning Suggestions for Large Networks</td>
</tr>
</tbody>
</table>
APPENDIX F  PROCESS PARAMETER FILE FORMAT

F.1 PARAMETER SECTION
F.1.1 IDENT  F-2
F.1.2 LOWTHRESH and HIGHTHRESH  F-2
F.1.3 DEFAULT_TIMEOUT  F-3
F.1.4 MAXRES  F-3
F.1.5 Nonparasitic Capacitance Coefficients and Controls  F-3
F.1.6 Calculating the Node Capacitance  F-4

F.2 PARASITIC_PARAMETER
F.2.1 Area and Periphery Coefficients  F-5
F.2.2 Parasitic Capacitance Coefficients  F-7

F.3 RESISTANCE SECTION
F.3.1 Interpolation of Resistance Values  F-7
F.3.2 RESISTANCE_SECTION Syntax  F-8

F.4 DEVICE SECTION
F.4.1 DEVICE_SECTION Syntax  F-9
F.4.2 Parasitic-Parameter-Type Name  F-10
F.4.3 Case Keywords  F-10
F.4.4 DECSIM Primitive MOS Model Types  F-11
F.4.5 Resistance-Set-Names  F-11
F.4.6 Usage Names  F-11

APPENDIX G  BINARY TRACE FILE FORMAT

G.1 CREATING BINARY TRACE FILES
G.1.1 Optional and Required Qualifiers on the TRACE Command  G-1
G.1.2 Opening a Binary Trace File  G-2
G.1.3 Concatenating Related Traces into a Single Trace File  G-2

G.2 READING BINARY TRACE FILES
G.2.1 Module Header Records (MHR)  G-2
G.2.1.1 Main Module Header (MMH) • G-5
G.2.1.2 Module Description Record (MDR) • G-6
<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-2</td>
<td>Truth Table for a 2-input WIREAND Phantom</td>
</tr>
<tr>
<td>3-3</td>
<td>Truth Table for a 2-input WIREOR Phantom</td>
</tr>
<tr>
<td>3-4</td>
<td>Truth for TRISTATE or WIREMOS</td>
</tr>
<tr>
<td>3-5</td>
<td>Parameter Types That Can Be Specified</td>
</tr>
<tr>
<td>3-6</td>
<td>Primitive Parameters That Can Be Passed</td>
</tr>
<tr>
<td>3-7</td>
<td>Interpolating Min, Nom, and Max Values</td>
</tr>
<tr>
<td>4-1</td>
<td>Enable Signal Check Points</td>
</tr>
<tr>
<td>5-1</td>
<td>Pulselength and Cyclewidth Checker Qualifiers</td>
</tr>
<tr>
<td>5-2</td>
<td>Setup and Hold Checker Qualifiers</td>
</tr>
<tr>
<td>5-3</td>
<td>Conditional Checker Qualifiers</td>
</tr>
<tr>
<td>5-4</td>
<td>Routing of output from PRINT/[NO]LOG</td>
</tr>
<tr>
<td>5-5</td>
<td>Running DECSIM with AXE</td>
</tr>
<tr>
<td>5-6</td>
<td>TRACE Formats</td>
</tr>
<tr>
<td>5-7</td>
<td>TRACE Formats and Signal Types</td>
</tr>
<tr>
<td>5-8</td>
<td>TRACE Execution Qualifiers</td>
</tr>
<tr>
<td>5-9</td>
<td>TRACE Format Qualifiers</td>
</tr>
<tr>
<td>A-1</td>
<td>Command and Variant Prefix Summary</td>
</tr>
<tr>
<td>B-1</td>
<td>Nonprinting Control Characters</td>
</tr>
<tr>
<td>B-2</td>
<td>Printing Characters</td>
</tr>
<tr>
<td>C-1</td>
<td>DECSIM Symbols and their Character Sets</td>
</tr>
<tr>
<td>D-1</td>
<td>Expression Operators by Relative Precedence</td>
</tr>
<tr>
<td>D-2</td>
<td>Directives</td>
</tr>
<tr>
<td>D-3</td>
<td>Wildcard Applicability</td>
</tr>
<tr>
<td>D-4</td>
<td>Shift Operator Definitions</td>
</tr>
<tr>
<td>D-5</td>
<td>TS Truth Table</td>
</tr>
<tr>
<td>D-6</td>
<td>MOSTS Truth Table</td>
</tr>
<tr>
<td>D-7</td>
<td>TSCON Truth Table</td>
</tr>
</tbody>
</table>

3-32
3-33
3-34
3-48
3-50
3-52
4-54
5-99
5-99
5-100
5-166
5-185
5-221
5-221
5-222
5-222
A-1
B-1
B-1
C-2
D-2
D-4
D-5
D-34
D-36
D-37
D-37
Preface

Definition of DECSIM

DECSIM provides languages that enable you to model, simulate, test, and debug multilevel logic networks, ranging from simple gates to whole CPUs. These are the three basic steps you follow in using DECSIM.

1 Construct models at the functional, gate, or MOS level, with the appropriate modeling language: the behavior language or the interconnect language.

2 Compile and load the models into the DECSIM simulator, where you can debug them interactively with the interactive command language.

3 Simulate your models, using the interactive command language.

This manual contains the information you need to complete all three steps.

Goals of the Manual

The DECSIM Reference Manual provides detailed descriptions of the syntax and functionality used by the DECSIM Logic Simulator. Use this manual if you are generally familiar with DECSIM and want specific information—for example, information on a command, qualifier, or how primitives are defined and interconnected.

Document Structure

The DECSIM Reference Manual is divided into two parts.

Part I contains the information needed to construct models.

Chapter 1 describes three primitives (equation, memory, and MOS) and hazard checkers.

Chapter 2 describes the behavior primitive, which is also a model.

Chapter 3 describes the conventions that are used both for interconnecting primitives and for interconnecting models constructed of primitives.

Part II contains the information you need to compile and simulate models: conventions, commands, and qualifiers.

Chapter 4 describes simulation conventions such as character sets, signal accessing, escape prompting, and fault simulation.

Chapter 5 lists the commands, with their qualifiers, in alphabetical order.
Intended Audience

To use this manual, you should have a basic understanding of DECSIM and a knowledge of chip design.

Where to Get Additional Help

The DECSIM documentation set includes:

- *DECSIM User's Guide* (the process of constructing and simulating models)
- *DECSIM Reference Manual* (the separate entities of DECSIM)
- *DECSIM Quick Reference Guide* (syntax and brief descriptions of commands)

On-line help includes:

- DECSIM$SYS:DECSIM.DOC (DECSIM release notes)
- DECSIM$SYS:ANINDEX.TXT (index of application notes)
- Help command (information on commands and qualifiers)
- Escape prompting (fields and options for completing commands)

Training is also available. Contact DECSIM::DECSIM_SUPPORT for schedules.

Conventions

- Brackets ([ ] ) indicate optional phrases.
- Brackets containing a delimiter such as a comma, followed by three dots ( [,...] ), indicate that you can repeat optional syntax by separating each item with the delimiter.
- Text in uppercase (for example, RADIX) indicates a keyword.
- Text in lowercase indicates a generic field for you to enter. For example, LOG/qualifier can indicate LOG/NORESPONSES or LOG/COMMANDS.
- All punctuation characters except brackets and ellipses must be typed as shown.
- In interactive examples, the user's entries are in boldface type and DECSIM's responses are in regular type.
Part 1  Constructing and Connecting Models
1

How Primitives Are Defined

This chapter describes the smallest pieces of the models—the primitives. These include equation statements, MOS primitives, and memory primitives. This chapter also covers hazard checkers, which can be instantiated in models like primitives. A behavior primitive, which consists of an entire behavior model, is defined in Chapter 2.

Chapter 3, Interconnect Language, describes how to assemble the primitives into models and how to assemble those models hierarchically into networks.

Together, primitives and the interconnect language comprise the elements you use to describe your network.

1.1

EQUATION STATEMENTS

1.1.1 Environment

Equation statements are contained in models. The following "NOT" gate (inverter) shows the relationship of an equation primitive to a model.

MODEL NOT [RISE=10, FALL=10] 1/OUT = 2/IN;
EQUATION [RISE0_1 = RISE, FALL1_0 = FALL] OUT = NOT IN; !Primitive
ENDMODEL NOT;

1.1.2 Definition

The equation statement defines a primitive whose functionality is specified with a Boolean-like expression.

The equation statement has one output and from one to six inputs. All inputs and outputs are 4-state (0, 1, U, Z) and must be exactly one bit wide.

Format

EQUATION [[parameters]] outputname = equation expression;

EQN [[parameters]] outputname = equation expression;

Example

eqn [ rise0_1 = 10, fall1_0 = 10 ] out = .in1 and .in2 ;

Models of commonly used logic, such as inverters, gates, and flip-flops, are constructed from equation primitives and stored in model library files, such as the default library GATES.NET. For a complete list of the libraries currently supported by DECSIM, see Section 1.1.6.
1.1.3 Components

The following sections describe the components of a DECSIM equation statement.

1.1.3.1 Keyword

The keyword EQUATION, which may be represented as EQN, indicates that what follows is an equation statement.

1.1.3.2 Parameters

There are two ways to specify RISE-FALL propagation delays—either explicitly in the EQUATION statement, or by parameter passing. See Section 3.8 for more information on parameter passing.

The equation parameters describe the propagation delay for each of the 12 possible output state transitions. There are twelve parameter keywords for these state transitions (six types of rise transitions, and six types of fall transitions). For example, RISE0_1 specifies a rise from zero to one.

NETPRO, one of the DECSIM compilers, calculates the propagation delays of those parameter types not specified. The delay defaults to 0 if no parameters at all are specified.

The parameters appear in a list within square brackets ([ ]). The parameter list can be either name-based or order-based, as described below; however, these two forms cannot be mixed.

Name-based Equation Parameters

A name-based equation parameter list consists of one or more parameter types that can be specified in any order.

Format

EQUATION [ parameter keyword = parameter triple [ , ... ] ]

Example

EQUATION [RISE0_1 = 10, FALL1_0 = 10]

The valid name-based parameter keywords are:

RISE0_1, FALL1_0
RISE0_Z, FALLZ_0
RIZE1_1, FALL1_Z
RISE0_U, FALLU_0
RISEU_1, FALL1_U
RISEU_Z, FALLZ_U

The parameter triple consists of three delay times (min:nom:max), where min stands for minimum, nom for nominal, and max for maximum.

The max time must be greater than or equal to the nom time, which must be greater than or equal to the min time, which must be greater than or equal to zero.

If you specify only one value for time, that value is used for min, nom, and max.
How Primitives Are Defined

If you specify two values, NETPRO calculates the nom value according to the formula:

\[
\text{nom} = \frac{\text{min} + \text{max}}{2}.
\]

If you do not specify a parameter type, NETPRO calculates the value from those you have specified.

Example

\[
\text{[RISE}_0\text{.1 = 1NS, FALL}_1\text{.0 = 2NS, RISE}_1\text{.Z = 3NS, FALL}_1\text{.Z = 4NS]}
\]

The parameter types shown here are assigned their respective values. The rest of the values for the parameters not shown are calculated as described below.

Order-based Equation Parameters

Order-based equation parameters form a list of parameter types which are specified not by name, but by order. They are drawn from the list in the following order:

RISE\_0\_1
FALL\_1\_0
RISE\_0\_Z
FALL\_0\_0
RISE\_1
FALL\_1\_Z
RISE\_0\_U
FALL\_U\_0
RISE\_U\_1
FALL\_U\_U
RISE\_U
FALL\_Z

As in name-based parameters, each type in the list takes a parameter triple. You can omit a parameter type from an order-based list by leaving a comma to mark its place. You can omit parameter types from the end of the list without using commas.

Example

\[
\text{[1NS, 2NS, , , 3NS, 4NS]}
\]

In the example, parameter values are assigned in the following manner:
RISE\_0\_1 is set to 1NS, FALL\_1\_0 is set to 2NS, RISE\_1 is set to 3NS, FALL\_1\_Z is set to 4NS. NETPRO calculates the rest of the parameters.

Calculating Unspecified Equation Parameters

The procedure for calculating values for unspecified equation parameters is described after Table 1–1.
Table 1–1  Categories of Parameters

<table>
<thead>
<tr>
<th>Logic State</th>
<th>0</th>
<th>1</th>
<th>Z</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>xx</td>
<td>RISE0_1</td>
<td>RISE0_Z</td>
<td>RISE0_U</td>
</tr>
<tr>
<td>1</td>
<td>FALL1_0</td>
<td>xx</td>
<td>FALL1_Z</td>
<td>FALL1_U</td>
</tr>
<tr>
<td>Z</td>
<td>FALLZ_0</td>
<td>RISEZ_1</td>
<td>xx</td>
<td>FALLZ_U</td>
</tr>
<tr>
<td>U</td>
<td>FALLU_0</td>
<td>RISEU_1</td>
<td>RISEU_Z</td>
<td>xx</td>
</tr>
</tbody>
</table>

Formula for Calculating Unspecified Parameters

First, NETPRO calculates any values for missing parameter types listed inside the box in the above table, as follows:

1. The six parameter types inside the box are split into three pairs, identified as P1, P2, P3:
   - RISE0_1 and FALL1_0 are P1
   - RISEZ_1 and FALLZ_0 are P2
   - RISE0_Z and FALL1_Z are P3

2. If a pair has only one value specified, NETPRO sets both values of that pair to the specified value.

3. Now all pairs are either full (i.e., both parameter values specified) or empty (i.e. no parameter values specified). All three pairs must now be filled (see Table 1–2 below).

Table 1–2  Values Assigned to Parameter Pairs (P1, P2, P3)

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>Empty</td>
<td>Empty</td>
<td>P1, P2, and P3 are assigned values of 0.</td>
</tr>
<tr>
<td>Empty</td>
<td>Empty</td>
<td>Full</td>
<td>P1 and P2 are assigned the value of P3.</td>
</tr>
<tr>
<td>Empty</td>
<td>Full</td>
<td>Empty</td>
<td>P3 and P1 are assigned the value of P2.</td>
</tr>
<tr>
<td>Empty</td>
<td>Full</td>
<td>Full</td>
<td>P1 is assigned the value of P2.</td>
</tr>
<tr>
<td>Full</td>
<td>Empty</td>
<td>Empty</td>
<td>P2 and P3 are assigned the value of P1.</td>
</tr>
<tr>
<td>Full</td>
<td>Empty</td>
<td>Full</td>
<td>P2 is assigned the value of P1.</td>
</tr>
<tr>
<td>Full</td>
<td>Full</td>
<td>Empty</td>
<td>P3 is assigned the value of P2.</td>
</tr>
</tbody>
</table>

Second, NETPRO calculates the U parameters from the parameter types inside the box, according to the table below. The U parameter determines what states are valid.
### Table 1–3 Calculating Unspecified Parameters

<table>
<thead>
<tr>
<th>U Parameter</th>
<th>Valid States</th>
<th>U Parameter Set To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALLU_0</td>
<td>1</td>
<td>FALL1_0</td>
</tr>
<tr>
<td></td>
<td>Z</td>
<td>FALLZ_0</td>
</tr>
<tr>
<td></td>
<td>not Z, not 1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Z and 1</td>
<td>MAX (FALL1_0, FALLZ_0)</td>
</tr>
<tr>
<td>RISEU_1</td>
<td>0</td>
<td>RISE0_1</td>
</tr>
<tr>
<td></td>
<td>Z</td>
<td>RISEZ_1</td>
</tr>
<tr>
<td></td>
<td>not 0, not Z</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 and Z</td>
<td>MAX (RISE0_1, RISEZ_1)</td>
</tr>
<tr>
<td>RISEU_Z</td>
<td>0</td>
<td>RISE0_Z</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>FALL1_Z</td>
</tr>
<tr>
<td></td>
<td>not 0, not 1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 and 1</td>
<td>MAX (RISE0_Z, FALL1_Z)</td>
</tr>
<tr>
<td>RISEU_U</td>
<td>1</td>
<td>RISE0_1</td>
</tr>
<tr>
<td></td>
<td>Z</td>
<td>RISE0_Z</td>
</tr>
<tr>
<td></td>
<td>not 1, not Z</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1 and Z</td>
<td>MIN (RISE0_1, RISE0_Z)</td>
</tr>
<tr>
<td>FALL1_U</td>
<td>0</td>
<td>FALL1_0</td>
</tr>
<tr>
<td></td>
<td>Z</td>
<td>FALL1_Z</td>
</tr>
<tr>
<td></td>
<td>not 0, not Z</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 and Z</td>
<td>MIN (FALL1_0, FALL1_Z)</td>
</tr>
<tr>
<td>FALLZ_U</td>
<td>0</td>
<td>FALLZ_0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>RISEZ_1</td>
</tr>
<tr>
<td></td>
<td>not 0, not 1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 and 1</td>
<td>MIN (FALLZ_0, RISEZ_1)</td>
</tr>
</tbody>
</table>

**Note:** Any equation parameter type explicitly specified for an invalid state results in a warning.

### 1.1.3.3 Output Name

The equation output name must be one bit wide.

**Example**

\[ \text{EQUATION OUT} = A \text{ AND } B; \]

### 1.1.3.4 Equation Expression

The equation expression is the heart of the equation statement. It describes the behavior or function of the statement in terms of its inputs, as well as indicating which signals are inputs to the equation.

The expression is made up of operators and operands, which are described in the following sections.

**Operands**

There are two types of equation expression operands: constants and input signal names. Constants provide numeric values.
How Primitives Are Defined

DECSIM allows great flexibility in defining signal names; thus, operators, constants, and input names can look alike. The following sections explain how DECSIM distinguishes one from another.

Constant Operands

Digits are numeric values unless they are enclosed in quotation marks, in which case they are names.

Example

\[
\text{EQUATION } 0 = I \text{ AND } 1; \quad !\text{DECSIM interprets "1" as a constant}
\]

\[
\text{EQUATION } 0 = I \text{ AND } "1"; \quad !\text{DECSIM interprets "1" as a signal name}
\]

The valid constants are 0, 1, Z#2, and U#2. The logic values U and Z must each be followed by #2; otherwise NETPRO interprets them as signal names.

Input Operands

Input operands are signal names for single-bit wires. The following rules govern the use of quotation marks for names containing characters other than letters and digits. Such names can look like operators.

- Signal names that begin with a letter and contain only letters and digits need not be enclosed in quotation marks.

- A signal name must be enclosed in quotation marks if it has embedded spaces or contains the following standard signal name characters:
  \[ + - ( ) \]

- If a signal name contains special characters not normally allowed, those characters must be enclosed in quotation marks.

Note: Because the interactive command language interprets a dot (.), an asterisk (*), and a question mark (?) as special characters, whether they are enclosed in quotation marks or not, it is recommended that you do not use them in names.

- If a signal name can be interpreted as a keyword, operator, macro, or numerical constant, it must be enclosed in quotation marks.

- If the double quotation mark character is part of a signal name, the entire name, including double quotation marks, must be enclosed in quotation marks.

Examples

\[
\text{EQUATION } [10,10] \text{ OUT } = \text{ "NAME AND SPACE" OR NAMEWITHOUTSPACE } ;
\]

\[
\text{EQUATION } [5:10,4:8] \text{ XYZY } = \text{ A AND (B OR "XOR") } ;
\]

Operators

Operator expressions specify the operation to be performed on expression operands. The following operators are recognized within equation statements.
1.1.4 Feedback in Equations

DECSIM supports feedback in equation statements; that is, DECSIM can use the expression's output at the time of evaluation as part of the equation expression. The following is an example of an RS flip-flop modeled with one equation statement:

```
EQUATION q = r NAND (s NAND q);
```

The output in the equation is evaluated once per input transition. "q" is not considered to be an input. If f(q) does not equal f(f(q)), NETPRO issues an "unstable" warning and evaluates only f(q). Thus, the actual output of the equation is not seen.

Figure 1–1 shows the logic expressed in the above equation statement.

**Figure 1–1  RS Flip–Flop**

---

1.1.5 Restrictions

Restrictions on equation statements are as follows:

- Equations must not have more than six inputs. Even though an individual input appears more than once in an equation, it is counted as one input. For example, `EQN out = (a and s) or (b and not s);` has three inputs.

- Input and output signals must be one bit wide.

- Every equation must evaluate to U#2 (undefined) when all inputs are U#2. Violations often occur when constants appear in the equation or when the CONVU1 or CONVU0 operators are used. For example, `EQN out = CONVU1 a;` is invalid.
1.1.6 DECSIM Library Files

Structural models of commonly used logic such as inverters, NAND gates, and D-flips are contained in the default library file GATES.NET. Similar models for ZYCAD are contained in ZGATES.NET. In addition, there is a separate library that supports models which use feedback in equations, FFBGATES.NET. DCSMOS.LIB is a library file for MOS models.
1.2 MOS Primitives

1.2.1 Environment

There are two types of MOS primitives. They are defined by the MOSFET statement and the RESISTOR statement, and are both contained in models. The following two models, taken from a MOS library file, show the relationship of MOS primitives to a model.

```
MODEL MN [W=10, L=10] S,D=G;
BIDIRECTIONAL S,D;
MOSFET [TYPE='N', WIDTH=W, LENGTH=L] S,D=G; !MOSFET primitive
ENDMODEL MN;

MODEL RES [R=10000] R1,R2=;
BIDIRECTIONAL R1,R2;
RESISTOR [RESISTANCE=R] R1, R2=; !RESISTOR primitive
ENDMODEL RES;
```

Primitive models containing MOSFET and RESISTOR statements are generally kept in a MOS model library file, such as the default library DCSMOSLIB.NET.

1.2.2 Definition

The MOSFET statement defines a primitive whose functionality is specified by parameters (width, length, and type) and a connection list (source, drain, and gate).

The RESISTOR statement defines a primitive whose functionality is specified by a parameter (resistance) and a connection list (a connection on each side of the resistor).

**Format**

```
MOSFET [parameter-list] connection-list;

RESISTOR [resistance-parameter] resistor-connections;
```

1.2.3 Components of the MOSFET Statement

<table>
<thead>
<tr>
<th>1.2.3.1</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>The keyword MOSFET indicates to NETPRO that what follows is a MOSFET statement.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1.2.3.2</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>The MOSFET parameters describe the primitive MOSFET, as follows:</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>A string enclosed by single quotation marks indicating a device type that matches a device defined in the DEVICE_SECTION of the Process Parameter File</td>
</tr>
<tr>
<td>Width</td>
<td>Width of MOSFET</td>
</tr>
<tr>
<td>Length</td>
<td>Length of MOSFET</td>
</tr>
</tbody>
</table>

The default unit is microns.
How Primitives Are Defined

There are two ways to specify width and length: either explicitly in the MOSFET statement, or by parameter passing. See Section 3.8 for more information.

Example

MOSFET [TYPE='N', WIDTH=10 um, LENGTH=10um] !explicit specification
MODEL MN [W=10, L=10] S,D=G;
BIDIRECTIONAL S,D;
MOSFET [TYPE='N', WIDTH=W, LENGTH=L] S,D=G; !L and W are formal
ENDMODEL MN;

Name–based Parameters

A MOSFET name-based parameter indicates the kinds of parameters and their values.

Format

TYPE='type-string', WIDTH=distance, LENGTH=distance

Example

MOSFET [TYPE='N', WIDTH=10 um, LENGTH=20 um]
MOSFET [TYPE='N', WIDTH=foo, LENGTH=baz] !foo and baz are formal
parameters

Order–based Parameters

A MOSFET order-based parameter is a list of values arranged in the following order:

'type-string', width-distance, length-distance

Example

MOSFET ['N', 10 um, 20 um]
MOSFET ['N', foo, baz] !foo and baz are formal parameters

1.2.3.3 MOSFET Connections

The MOSFET connection list specifies names for the source, drain, and gate.

MOSFET Name–based Connections

MOSFET name-based connections indicate the type of signal and the signal name. The signal type is separated from the signal name by a slash (/). Signal–type/signal–name pairs are separated from each other by commas.

Example

SOURCE/source-signal, DRAIN/drain-signal, GATE/gate-signal;

MOSFET Order–based Connections
MOSFET order-based connections list the signal names in the following order:

source-signal, drain-signal = gate-signal;

The source and drain signals appear on the left side of the equal sign; the gate signal appears on the right side.

1.2.4 Components of the RESISTOR Statement

Format

RESISTOR [resistance-parameter] resistor-connections ;

1.2.4.4 Keyword

The keyword RESISTOR indicates to NETPRO that what follows is a RESISTOR statement.

1.2.4.5 Parameters

There are two ways to specify resistance—either explicitly in the RESISTOR statement, or by parameter passing. See Section 3.8 for more information on parameter passing. Units can be specified. The default unit is ohms. You can also use a decimal point and fractional values (for example, [R = 7.5]).

Example

RESISTOR [RESISTANCE=10000] ;

MODEL RES [R=10000] B1,B2=;

RESISTOR [RESISTANCE=R] B1, B2=;

ENDDMODEL RES;

Name-based Parameters

RESISTOR name-based parameters indicate the kind of parameter followed by an explicit value or a formal parameter.

Example

RESISTOR [RESISTANCE = 10K]

RESISTOR [RESISTANCE = FOO] !FOO is a formal parameter.

Order-based Parameters

RESISTOR order-based parameters specify an explicit value or a formal parameter without indicating the kind of parameter.

Example

RESISTOR [10K]

RESISTOR [FOO] !FOO is a formal parameter.
How Primitives Are Defined

1.2.6 Connections
RESISTOR connections are the two ends of the resistor. They are always order-based.

Order-based Connections
The connection-list for the RESISTOR statement contains two signal names to the left of the equal sign and none to the right.

Example

\[
s_{i1}, s_{i2} =
\]

1.2.5 MOSFET and RESISTOR Restrictions
The parameter “TYPE” must be a device defined in the Process Parameter File that is supplied with DECSIM, and that NETPRO uses when compiling the top-level network.

1.2.6 Length and Width Shrinkage
To specify the amount by which a MOS device shrinks during the fabrication process, you can indicate DELTA-L, the shrinkage in the length of the device, and DELTA-W, the shrinkage in the width, in the MOS Process Parameter File under the parasitic capacitance section. When DECSIM calculates the gate capacitance, periphery, and resistance values, DELTA-L and DELTA-W are subtracted from the length and width, respectively. By default, DELTA-L and DELTA-W are set to 0. See Appendix F for information on the MOS Process Parameter File.
1.3 MEMORY PRIMITIVES

1.3.1 Environment

Memory blocks are contained in models. The following example shows the relationship of a memory block to a model.

```
MODEL MEM_512_X_8
  DATA_OUT_SIG<7:0>, WRITE_ERROR_OUTPUT, READ_ERROR_OUTPUT =
  ADDR_SIG<8:0>, DATA_IN_SIG<7:0>, WRITE_ENABLE, READ_ENABLE;
STRUCTURE;
  MEMORY ARRAY_NAME (M1);
    WRITE_PORT [TCEN = 100]
      ADDRESS/ADDR_SIG<8:0>, DATAIN/DATA_IN_SIG<7:0>,
      CONTROL/WRITE_ENABLE, ERROR/WRITE_ERROR_OUTPUT;
    READ_PORT [TCEN = 100]
      ADDRESS/ADDR_SIG<8:0>, DATAOUT/DATA_OUT_SIG<7:0>,
      CONTROL/READ_ENABLE, ERROR/READ_ERROR_OUTPUT;
ENDMEMORY;
ENDSTRUCTURE;
ENDMODEL MEM_512_X_8;
```

1.3.2 Definition

The memory block defines a primitive whose functionality is that of a hardware memory and whose definition is specified with memory attributes, port parameters, and port connection lists.

Format

```
MEMORY [attributes];
  READ_PORT [parameters] connection list; !Multiple READ and WRITE
  WRITE_PORT [parameters] connection list; !ports are valid
ENDMEMORY;
```

There are two types of memories: ZYCAD edge–triggered (triggered by a change in the enable line) and level–sensitive (triggered either by the enable line going high, or by a change on any other input while the enable line is held high).

Level–sensitive memories are used only in soft (non–ZYCAD style) DECSIM.

ZYCAD edge–triggered memories are used in ZYCAD (hard DECSIM) simulations and in the soft DECSIM ZYCAD–style simulations.

**Note:** ZYCAD edge–triggering differs from classical edge–triggering. See the ZYCAD Reference Manual for more information.

By default, DECSIM memories are level–sensitive.
1.3.3 Components

1.3.3.1 MEMORY, ENDMEMORY Keywords
The keyword MEMORY indicates to DECSIM that what follows is a
memory block. The keyword ENDMEMORY signifies the completion of the
memory block.

1.3.3.2 Memory Attributes
Memory attributes are specified within the memory block following the
MEMORY keyword. They are optional (except for ARRAY_NAME), can
appear in any order, and are separated by spaces.

Memory attributes are specified with the following keywords and
arguments:

LEVEL_SENSITIVE
Specifies that the DECSIM simulation of the memory respond to the
state of the enable line. As long as the enable line is high, changes in
the address lines and data lines affect the memory. DECSIM simulates
LEVEL_SENSITIVE memories as FOURSTATE devices.

This attribute prohibits simulation by ZYCAD and by ZYCAD–mode
DECSIM. LEVEL_SENSITIVE is a default attribute.

ZYCAD_EDGE_TRIGGERED
Specifies that the DECSIM simulation of the memory be ZYCAD edge–
triggered. When the state of the enable line changes from 0 to 1, the
address line and data line are read. Once the enable line is high, changes
in the address lines and data lines do not affect the memory read or write
operation.

This attribute must be used with all ZYCAD edge–triggered memories,
whether hard ZYCAD or soft DECSIM.

MIN_ADDRESS(integer_value)
Specifies the lowest allocated address in the memory array. The default is
0.

MIN_ADDRESS is not supported by the DECSIM/ZYCAD interface. If
you use it with ZYCAD simulation, you receive warnings and the ZYCAD
network translation continues.

MAX_ADDRESS(integer_value)
Specifies the highest allocated address in the memory array. The default
is 2**W–1(exponential), where W is the number of address bits driving the
port.

MAX_ADDRESS is not supported by the DECSIM/ZYCAD interface; using
it with ZYCAD simulation causes errors.

INITIAL_STATE(simulation_state)
Specifies the initialized state of the memory (0, 1, or U#2) The default is
U#2 (undefined). The default for a 2–state memory is 0.
ARRAY_NAME(name)
Specifies a name for the memory array allocation, where you can load data from external files. You can specify its location in the network by using the same naming conventions that are used in creating hierarchical signal names.

U_STATE_RESOLUTION = integer_value
Specifies the number of U's (undefined states) that will be resolved in the address. This attribute is not fully implemented and does not allow you to specify a value. It uses only default values: eight for level-sensitive memories, zero for ZYCAD edge-triggered memories.

For ZYCAD_EDGE_TRIGGERED memories, U_STATE_RESOLUTION is always 0, and the default is 0.

FOURSTATE
A ZYCAD-specific attribute that specifies that each bit of the memory word can hold a 0, 1, or U. A Z is converted to a U for all memory inputs. This affects all ZYCAD edge-triggered memories, soft or hard. The default for all simulation modes is FOURSTATE.

TWOSTATE
A ZYCAD-specific attribute that specifies that each bit of the memory word can hold a 0 or a 1. This affects all ZYCAD edge-triggered memories, soft or hard. The default is FOURSTATE.

TWOSTATE is not available for LEVEL_SENSITIVE memories.

<table>
<thead>
<tr>
<th>1.3.3.3</th>
<th>Ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>There are two types of ports, READ and WRITE, specified with the READ_PORT and WRITE_PORT keywords.</td>
<td></td>
</tr>
</tbody>
</table>

READ_PORT Parameters
Parameters follow the READ_PORT keyword, appear in square brackets, and are separated by commas. Parameters include delays (TCER) and DISABLE_STATE. For level-sensitive memories, delays can be greater than 0. The default is 0.

For ZYCAD edge-triggered memories, the delay keyword (TCER) must specify 0 – the default. To change the delay of these memories, adjust the read line strobe.

The keywords are:

TCER=time_value_triple
Specifies the amount of time from the CONTROL input becoming enabled to the data read appearing at the output. The time_value_triple is one number that serves as the minimum, nominal, and maximum value. The default is 0 NS.

DISABLE_STATE(simulation_state), where simulation_state is 0,1,U#2 or Z#2
Specifies the state of the output when the enable input is 0. The default gives the output the state of the last read operation. DISABLE_STATE applies only to READ_PORTS associated with LEVEL_SENSITIVE memories.
READ_PORT Connections

A READ_PORT has four connections:

- ERROR—output connection, width is 1 bit, 4-state
- DATAOUT—output connection, width is 1-256 bits, 4-state
- CONTROL—read enable input connection, width is 1 bit, 4-state
- ADDRESS—input connection, width is 1-256 bits, 4-state

READ_PORT connections can be name-based or order-based.

A READ_PORT name-based connection indicates the kind of connection and its value. These connections can appear in any order.

Format

ADDRESS/address_signal, DATAOUT/output_signal,
CONTROL/control_signal,
ERROR/error_signal

READ_PORT order-based connections must be ordered as follows:

data-out, error = address, control;

WRITE_PORT Parameters

There is one WRITE_PORT parameter. It is:

TCEW=time_value_triple

Specifies the time from the CONTROL input becoming enabled to the data being written from the DATAIN input into the internal memory array. No delays are allowed in ZYCAD edge-triggered memories. The default for all modes is 0.

WRITE_PORT Connections

A WRITE_PORT has four connections:

- ERROR—output connection, width is 1 bit, 4-state
- CONTROL—input connection, width is 1 bit, 4-state
- ADDRESS—input connection, width is 1-256 bits, 4-state
- DATAIN—input connection, width is 1-256 bits, 4-state

WRITE_PORT connections can be name-based or order-based.

A WRITE_PORT name-based connection indicates the kind of connection and its value. These connections can appear in any order.

Format

ADDRESS/address_signal, DATAIN/data_input_signal,
CONTROL/control_signal,
ERROR/error_signal
WRITE_PORT order–based connections must be ordered as follows:

    error = data-in, address, control;

1.3.4 Undefined Bits in Memory

Undefined bits are produced and handled differently in level–sensitive soft DECSIM than in ZYCAD.

1.3.4.4 Undefined Memory Bits in Soft DECSIM

- On a read, where any address bits are undefined, DECSIM expands the U to 1 and 0. It then does a bit–by–bit comparison of the data in the two locations. If the values are the same, it reads the data; if the values differ, it reads a U#2. If there are more than eight undefined bits, the ERROR bit is set.

- On a write, where any address bits are undefined, DECSIM expands the U to 1 and 0. It then does bit–by–bit comparisons of the DATAIN value with the contents of both memory locations; if the DATAIN value is the same as the contents of the memory location, it writes the value; if the values differ, it writes a U#2. If there are more than eight undefined bits, the ERROR bit is set.

- On a write, where any data bits are undefined, DECSIM inserts a U#2. Edge–triggered memories treat undefined bits the same way ZYCAD does.

1.3.4.5 Undefined Memory Bits in ZYCAD

In ZYCAD, undefined bits in memory produce errors and can be caused by any of three conditions:

- On a read, if any address bits are undefined, an error is produced and the data_out line contains all U's. The ERROR bit is set and the system variable %Z_MEMORY_ERRORS is incremented.

- On a write, if any address bits are undefined, an error is produced and the write is not performed. The ERROR bit is set and the system variable %Z_MEMORY_ERRORS is incremented.

- On a write, if any data bits are undefined and memory is twostate, an error is produced, and the write is not performed. The ERROR bit is set and the system variable %Z_MEMORY_ERRORS is incremented.
1.4 HAZARD CHECKERS

Hazard checkers enable you to detect time and data violations on scalar signals in equation, MOS, and memory models during simulation. The types of hazard checkers you can set are pulsewidth, cyclewidth, setup, and hold. You can also define any of these checkers as conditional checkers. These checkers and the types of violations they report are described in detail in Section 4.5.3.4. For a general discussion of hazard analysis and its role in simulation, see Section 4.5.3.

You can set hazard checkers interactively with the [SET] HAZARD command or by instantiating hazard checkers in a structural model. For information on defining checkers interactively, see the [SET] HAZARD command. See also the HAZARD command variants: CANCEL/DISABLE/ENABLE HAZARD and SHOW HAZARD.

1.4.1 Environment

Hazard checkers are not primitives, strictly speaking; they are simulation artifacts, like watches and detects. However, unlike other simulation artifacts, hazard checkers can be instantiated in a structural model. They never become part of the permanent network topology, but when the model is compiled or loaded, hazard checker definitions are passed to DECSIM.

The following model contains two hazard checkers and shows the relationship of the checkers to a model.

```
MODEL DFF_PC [r=10, f=10, s=16] q, q, = addr1, clk, pre, c;
STRUCTURE gates;
UNCONNECTED_OK q, q;
NAND3 [0,0] I1 = pre, I2, I4;
NAND3 [1,1] I2 = c, I1, clk;
NAND3 [0,0] I3 = I2, clk, I4;
NAND3 [1,1] I4 = I3, c, addr1;
NAND3 [r,f] q = pre, I2, q;
NAND3 [r,f] q = I3, c, q;
```

1. This is a cyclewidth checker without an optional user-defined label.
2. This is a setup checker with a user-defined label.

In the example above, the hazard checkers were defined at the lowest level of hierarchy in order to avoid having to redefine them each time the model DFF_PC is instantiated. However, it is possible to define hazard checkers at any level in the hierarchy.
Restrictions

There are a number of restrictions concerning hazard checker definitions in models:

- The checkers must be placed between MODEL and ENDMODEL declarations.
- The checkers must be placed between STRUCTURE and ENDS STRUCTURE declarations, if these declarations are specified.
- A model cannot consist of hazard definitions alone.
- The signals specified in the checkers must refer to scalar signals that are described within the same MODEL and ENDMODEL declaration.
- Hazard checkers cannot be placed on vector signals (behavioral or memory).
- Each user-defined label in the STRUCTURE/ENDSTRUCTURE block must be unique.

For a list of restrictions that apply to simulating with hazard checkers in your model, see Section 1.4.5.

1.4.2 Definition

A hazard checker sets a timing violation check on a data signal (also called a primary signal). A hazard checker defined in a structural model has the following format.

Format

label: %hazard_type [parameter_list] signal_list

The label is used to reference the hazard checker from the command language. It is optional; if none is specified, NETPRO assigns each checker a unique label. The format of the default is $\text{number}$, the same format as the labels NETPRO generates for model instantiations.

For example, in the model described in Section 1.4.1, NETPRO would generate a label in the $\text{number}$ format for each of the six NAND gates and for the cyclewidth checker, starting with $\text{50}$. The label for the cyclewidth checker would thus be $\text{56}$. The setup checker has a user-defined label, setup_addr1. When the model is instantiated, the checker is identified by a unique hazard name, which is the hazard label prefixed by the hierarchical model name, for example, A1.B1.$\text{56}$ and A1.B2.setup_addr1.

The %hazard_type must be one of the following keywords:

- %PULSEWIDTH
- %CYCLEWIDTH
- %SETUP
- %HOLD
The parameter list is a name-based list of parameters, separated by commas and enclosed in square brackets, that defines the functionality of the checker. The parameters that are required or optional vary with the type of checker and are described in Section 1.4.3 below.

The signal list is a name-based or order-based list of signals to which the checker applies. A name-based list has the following format:

\[
\text{signal_type/ signal_name [, signal_type/ signal_name, ...]}
\]

where signal_type is one of the following keywords:

- DATA
- REFERENCE
- ENABLE

An order-based list has the following format:

\[
\text{signal1 [, signal2, signal3]}
\]

where signal1 is the data signal, signal2 is the reference signal (in setup and hold checkers), and signal3 is the enable signal (in conditional checkers). The number and types of signals required in the signal list vary with the type of checker and are described in Section 1.4.3.

Note: Only one signal of each type may be specified; the wildcard * is not allowed.

1.4.3 Hazard Checker Parameters

This section describes the various types of hazard checkers and their parameters. The parameter fields are defined below.

- **state**—One of the four DECSIM states (0, 1, U, Z). The #2 specification is not allowed in this context. Wildcards are not allowed.

- **edge**—A single transition expressed as \( \text{state}_{\text{state}} \), where \text{state} is one of the four DECSIM states. The #2 specification is not allowed in this context. Wildcards are not allowed.

- **state_list**—A single state or a list of states separated by commas and enclosed in parentheses. The wildcard X may be used. Thus \( \text{DATA}_{\text{STATE}}=X \) is equivalent to \( \text{DATA}_{\text{STATE}}=(1, 0, U, Z) \).

- **edge_list**—A single edge or a list of edges separated by commas and enclosed in parentheses. The wildcard X may be used to represent one or both states in a transition. Thus \( \text{REFERENCE}_{\text{EDGE}}=X \_1 \) is equivalent to \( \text{REFERENCE}_{\text{EDGE}}=(U \_1, Z \_1, 0 \_1) \), and \( \text{REFERENCE}_{\text{EDGE}}=X \_X \) means any transition on the reference signal.

- **time**—A single time value, with or without time units, or a model parameter name. The default time unit is nanoseconds.
1.4.3.1 Pulsewidth and Cylewidth Checkers

Pulsewidth and cyclewidth checkers check the waveform of a repeating signal, such as a clock, during a window of time specified by the user. A pulsewidth is defined as a transition to and from a specified state. A cyclewidth is a transition to a specified state, a transition to another state or states, and a transition back to the specified state.

The following parameters can be used in pulsewidth and cyclewidth hazard checkers.

- \texttt{MAX\_WIDTH=\text{time}}—specifies the maximum duration of the pulsewidth or cyclewidth.
- \texttt{MIN\_WIDTH=\text{time}}—specifies the minimum duration of the pulsewidth or cyclewidth.
- \texttt{DATA\_STATE=\text{state}}—specifies the level at which the duration of the pulsewidth or cyclewidth is checked.
- \texttt{DATA\_EDGE=\text{edge}}—specifies the edge at which the signal is checked to see if the waveform is square. For example, \texttt{DATA\_EDGE=0\_1} in a pulsewidth checker would check for a square pulse from 0 to 1 and back to 0.

Defaults and Restrictions

You can use both the \texttt{MAX\_WIDTH} and the \texttt{MIN\_WIDTH} parameters in a pulsewidth or cyclewidth checker to define the window during which the waveform is checked; you must use at least one. You cannot use both the \texttt{DATA\_EDGE} and the \texttt{DATA\_STATE} parameters in a single checker.

You must specify a single \texttt{DATA} signal in the \texttt{signal\_list}. You may also specify an \texttt{ENABLE} signal (see Section 1.4.3.3). The order of signals in an order-based signal list is: \texttt{data signal [, enable signal]}.

Default: \texttt{DATA\_STATE=1} (level check)

Examples

```
\$PULSEWIDTH [MIN\_WIDTH = 40, MAX\_WIDTH = 70] DATA/clock;

Pulse3: \$PULSEWIDTH [MIN\_WIDTH = 9, MAX\_WIDTH = 20, DATA\_EDGE = 1\_0] DATA/strobe0;

\$CYCLEWIDTH [MAX\_WIDTH = 25, DATA\_EDGE = 0\_1] DATA/strobe1;

Cycle3: \$CYCLEWIDTH [MIN\_WIDTH = 10, MAX\_WIDTH = 25] DATA/strobe0;
```

1–21
How Primitives Are Defined

1.4.3.2 Setup and Hold Checkers

A setup checker defines a minimum time window during which the data signal must be stable before an edge on the reference signal. The hold checker defines a minimum time window during which the data signal must be stable beginning with an edge on the reference signal. A hold checker is symmetrically opposite to a setup checker, relative to the reference edge.

The following parameters can be used in setup and hold checkers:

- **WINDOW=time**—defines the minimum window during which the data signal must be stable before an edge on the reference signal (setup checkers) or beginning with that edge (hold checkers).
- **REFERENCE_EDGE=edge_list**—specifies the transition on the reference signal that ends the setup window or begins the hold window.
- **DATA_STATE=state_list**—identifies the state or states at which the data signal must be stable during the setup or hold window.
- **DATA_OFFSET=time**—offsets the setup or hold window relative to the reference edge. If a transition occurs within the specified offset, it is not reported as a violation.

Defaults and Restrictions

You must use the WINDOW=time parameter. Both the primary (data) signal and the reference signal must be specified. You may also specify an enable signal (see Section 1.4.3.3). The order of signals in an order-based signal list is: data signal, reference signal [, enable signal].

Defaults: **REFERENCE_EDGE=X_X**, **DATA_STATE=X**

Examples

```latex
\%SETUP: [WINDOW = 10, 
REFERENCE_EDGE = 1_0, 
DATA_STATE = (0,1)] DATA/ data1, REFERENCE/ clock;

\%SETUP [WINDOW=5, 
DATA_OFFSET=5 ] DATA/ data2, REFERENCE/ clock;

\%HOLD [WINDOW = 100ps, 
REFERENCE_EDGE = 1_U, 
DATA_STATE = 1 ] DATA/d0, REFERENCE/strobe0;

\%HOLD [WINDOW=6, 
REFERENCE_EDGE = (0_1, 2_1), 
DATA_STATE=(1,0), 
DATA_OFFSET=4 ] DATA/d1, REFERENCE/strobe1;
```

1.4.3.3 Conditional Hazard Checkers

A conditional checker is one that is processed only if another signal is at a particular state or set of states when the checker is activated. Any of the four types of hazard checkers described above can be defined as a conditional checker by adding the name of the enable signal to the signal list.

**ENABLE_STATE=state_list**—is an optional parameter and specifies the state(s) of the enable signal when the checker is to be activated.
How Primitives Are Defined

Defaults and Restrictions

To make a checker conditional, you must specify an enable signal. The order of signals in an order-based signal list is: data signal, [reference signal], enable signal.

Default: ENABLE_STATE=1

Examples

%PULSEWIDTH [MIN_WIDTH = 50,
MAX_WIDTH = 100,
DATA_STATE = 1,
ENABLE_STATE = 0 ] DATA/clock, ENABLE/enable;

1.4.4 NETPRO Checker Violation Reports

NETPRO hazard checkers report time and data violations in the FULL format. This format is described under [SET] HAZARD/REPORT in Chapter 5. The report includes the type of violation, the time it occurred, the signal name(s), and the expected and actual times of the transition. NETPRO checkers do not report successes.

Note: To capture NETPRO checker violation reports in a log file, you must execute a [SET] LOG/RESPONSES command before loading or compiling your model. Use the /HAZARD qualifier on the [SET] LOG command to put a .HAZ extension on the log file.

For more information on the violation reports and boundary conditions of these checkers, see Section 4.5.3.4.

1.4.5 Restrictions

In addition to the restrictions noted in Section 1.4.1, the following restrictions apply to simulating with NETPRO hazard checkers:

- Hazard checkers are not allowed on vector signals (behavioral or memory).
- Hazard checkers are ignored in ZYCAD networks and during fault simulation.
Behavior Primitives

The behavior primitive is a behavior model.

This chapter describes the contents of a behavior model, which is similar to a program in a high-level language.

Chapter 3, "Interconnect Language," describes how to connect a behavior model to other models.

2.1 Conventions

The following sections describe the conventions followed in the behavior language.

2.1.1 Character Set

The character set for behavior models comprises all printing ASCII characters (see Appendix B). No distinction is made between uppercase and lowercase. However, text strings, which are characters enclosed in single quotation marks (" "), are displayed exactly as they are typed, with the following exceptions:

- In text strings, an up-arrow is interpreted as a single ASCII control character, and must be followed by another character. For example, ^x is interpreted as "CTRL/X". Double up-arrows are interpreted as a single up-arrow. Consecutive single quotation marks are interpreted as one single quotation mark.

- Two consecutive up-arrows or consecutive single quotation marks in a text string print one up-arrow or one single quotation mark. For example, "" is displayed as ' , ^ is displayed as ^.

2.1.2 Comments

By preceding text with an exclamation point (!), you can write comments in your models that are ignored by the compilers. The compilers ignore text from the exclamation point to the end of the line.
2.1.3 Names

You create names for models, routines, state variables, synonyms, macros, modules, and labels. The following rules work for all names except module names. See Section 2.3.1 for more information about module names.

Names contain from 1 to 31 characters (with the exception of models, which contain 1 to 127 characters) created from the following character set:

- A–Z All uppercase letters
- a–z All lowercase letters (which DECSIM treats as uppercase)
- 0–9 All digits
- _ Underscore

The following restrictions are in force for names:

- Where names (such as 3 or A#16) can be interpreted as numbers, include some characters that are not numbers, or enclose the name in quotation marks (" "). See the following section on quoting.
- Names cannot contain spaces.
- Polarity has no meaning in behavior models, although you can indicate polarity in a name. See the following section on quoting.
- Names for port and state declarations must include subscripts to indicate the width, length (if an array), and numbering. See Section 2.1.6.

**Note:** Globally, you cannot declare the same name more than once, even if it has different subscripts. For example, you cannot specify both "input <0>" and "input <1>" as ports for the same model. Of course, input <1:0> is a valid 2-bit port.

The same rule applies to local names. However, you can declare a local name that is the same as a global name; but if you do, you can't access the global name.

- Using keywords as names is not recommended, but you can do it by enclosing them in quotation marks (" ").
- Do not begin names with the percentage sign (%) or dollar sign ($); they are reserved for DECSIM.

2.1.4 Quoting

By enclosing a name in quotation marks (" "), you can use any printing ASCII character (see Appendix B for a list of printing ASCII characters). If a name includes a nonstandard character, you must enclose the entire name in quotation marks.
**Behavior Primitives**

**Example**

"LONG MULTI-PART NAME"

"A + B"

The following restrictions are in force for quoting names:

- The number of characters within the quotation marks must be 31 or less. Names containing more than 31 characters are truncated with no warning message.

- If you enclose a number (a constant or a literal) in quotation marks, DECSIM interprets it as a name.

- To enclose a quotation mark in quotation marks, you must type two consecutive quotation marks. For example, "exam""ple" is interpreted as if there were one quotation mark between the m and the p.

- If quotation marks that enclose are part of the name (wirelisters sometimes create such names), you must enclose the name within double sets of quotation marks. For example, the name "x[3]" is expressed as """"x[3]"""". In such a case, word and bit subscripts are not interpreted, but are considered part of the name.

---

**2.1.5 Strings**

Strings are text enclosed in single quotation marks. Strings are part of the syntax for INFORM, MACRO, and %PRINT.

**Syntax**

'characters'

where characters can be any characters (A-Z, a-z, 0-9), or

- followed by a character (interpreted as a control key)
- followed by an ^ (interpreted as a single up-arrow (^))
- 'followed by a ' (interpreted as a single quotation mark ('))

To enclose a text string in single quotation marks, you must use consecutive single quotation marks. In the following example, the default parameter must be enclosed in single quotation marks. The parameter is a text string, which must also be enclosed in single quotation marks.

**Example**

MACRO error [errmsg = '''no error message'''] =

BEGIN
  %PRINT ('Error: ', errmsg);
  %PRINT ('pc = ', pc);
  HALT;
END
$ENDMACRO;
2.1.6 Word and Bit Subscripts

Word subscripts specify a word or range of words, and are enclosed in brackets ([ ]). Bit subscripts specify a bit or range of bits, and are enclosed in angle brackets (<>). The rules for subscripts in declarations differ from those for subscripts in statements or expressions.

2.1.6.1 Subscripts in Declarations

In declarations, subscripts specify the size of states, ports, or synonyms, and how the words and bits are numbered. Word and bit subscripts must be numbers or names declared as constants.

You can specify the range of a 1-bit declaration with a single number. For example, x<0> is equivalent to x<0:0>.

Syntax

- **Word Subscripts:** 
  \[ \text{num_or_name: num_or_name} \]
- **Bit Subscripts:** 
  \(< \text{num_or_name} >\)
  \(<\text{num_or_name: num_or_name}>\)
  \(<\text{num_or_name: num_or_name}>\)
  \(<\text{num_or_name: num_or_name}>\)
- **Used together:** 
  \([\text{word_subscripts}]<\text{bit_subscripts}>\)

Where:

- `num_or_name` is a number, or a name declared as a constant.
- A bit subscript with no values < defaults to the width and bit subscripts of the host machine. For example, the default on a VAX is 32 bits numbered <31:0>.

Example

```plaintext
STATE m[0:4095]<15:0>;  !word and bit subscript ranges
STATE b<2>;               !single bit subscript
```

The following restrictions are in force for subscripts in declarations:

- Word and bit subscripts must be numbers or names declared as constants; they cannot be expressions.
- Word subscripts must be a range; they cannot be just one word. Bit subscripts can be ranges or single bits.
- All word subscripts must be in ascending order. Bit subscripts can be in ascending or descending order.
- There is a limit of 256 bits per word. Bit subscripts must be in the range 0–255. Word subscripts must be in the range 0–FFFFFFF#16.
2.1.6.2 Subscripts in Statements and Expressions

In statements and expressions, subscripts can specify the particular subset of a state, port, or synonym that you want to access.

Word and bit subscripts can be any expression. Word subscripts must be one word; they cannot be a range of words. Bit subscripts can be ranges or single bits. If you don't specify bit subscripts, the default is the entire range. If a word subscript range has been declared, you must specify one word.

Syntax

Word Subscripts:

expression

Bit Subscripts:

< expression >

0: <expression:expression>

Used together:

[word_subscripts]<bit_subscripts>

where < expression > is a number, name declared as a constant, or operator expression

In the following example, the states without bit subscripts default to the declared width.

Example

psw<0> = c_vec<15>;  // bit subscript

mbr = m[mar];  // word subscript

mbr<7:0> = m[mar]<7:0>;  // word and bit subscripts

x = y[1 + 1];  // expression in subscript

a_sub_h<15:0> = a_lo<31:16>;  // range of bit subscripts

The assignment mbr = m[mar] is preferred over mbr<7:0> = m[mar]<7:0> when the entire range is being used.

Word and bit subscripts that are used or referenced must be consistent with those in the declaration. For all usages of a given name, the order must be consistent with that in the declaration. For example, if the order of the bit subscript range in the declaration is low to high (<0:15>), the order in the expression must also be low to high.

2.1.7 Connectives and Delimiters

The behavior language uses the comma (,) to separate like elements of a list, such as signal names or routine arguments.

The semicolon (;) separates statements and declarations.

Enclosing any character (such as a comma or semicolon) in quotation marks (" ") makes it a name.

The behavior compiler treats multiple spaces, tabs, and blank lines as one space and ignores those that aren't separators. However, extra spaces and tabs enclosed in single quotation marks (' ') are preserved.
Behavior Primitives

See Appendix B for a summary of how all ASCII characters are used in the DECSIM languages.

2.1.8 Numbers and Radixes

Numbers in the behavior language include whole numbers of radices 2–16. Decimal points are allowed only in time expressions.

Except for %PRINT, which has a default of radix 16, the default radix is 10.

Syntax

digits [$radix]

where digits are from the character set 0–9 and A–F (or a–f). Digits can include the characters U and Z (representing undefined and high impedance) for radices 2, 4, 8, and 16. U and Z in radices 4 and 8 expand to 2 and 3 bits of that value. However, U#16 is displayed as U.

Examples

1010#2
FFFF#16
U#2
Z#2
%PRINT (U#16);  !displays U

The width of a number is the amount of digits necessary to represent the number. It is not necessarily 32 bits. If the radix is 2, 4, 8, or 16, it includes any leading zeros; otherwise, leading zeros are ignored. For example, 0#2 is one bit; 00#2 is two bits.

Unary plus (+) and unary minus (−) increase the width of a number by one bit. For example, +1#2 is represented as 01#2; −1#2 is represented as 11#2.

known bug: %PRINT (radix = 2) (U#16) displays one U, not 4.

2.1.9 Statement Labels

You can prefix any statement (such as REPEAT, SELECT, or BEGIN–END) with a label. Unlike instance labels, statement labels are not visible from the command language—they exist solely as tags to be used in conjunction with LEAVE, specifying a statement from which to exit. See Section 2.6.12 for information on LEAVE.

Syntax

label_name: statement
Example

\[ x = 0; \]
\[ sloop: \text{REPEAT} \]
\[ \text{BEGIN} \]
\[ \text{IF memory}[x] \text{EQL value THEN LEAVE sloop; } \]
\[ x = x + 1; \]
\[ \text{END}; \]

The label can be any unique name. See Section 2.1.3 for more information on names.

A given label can be defined only once in each routine.

2.2 Model Definition

A behavior model is a high-level functional description of a network. It defines a primitive whose functionality is specified by procedural statements in BDS (the behavior language).

Behavior models can have multiple inputs and outputs, each with a width of 1–256 bits.

Output ports are always fourstate. Input ports can be either fourstate (1,0,U#2,Z#2) or twostate (1,0). Two-state is the default for input ports. Bidirectional ports are fourstate.

A behavior model source file (default extension .BDS) and a network description file (.NET extension) can contain only one behavior model.

Syntax

\[
\text{MODEL model_name [delay_parameters] [outputs] = [inputs];} \\
\text{BEHAVIOR [subblock_name] [modifiers] [declarations];} \\
\text{ENDBEHAVIOR [subblock_name];} \\
\text{ENDMODEL model_name;}
\]

The following example shows a .BDS file for a model of an inverter.

Example

\[
\text{MODEL invert [pl=3] } b<0> = a<0>; \\
\text{BEHAVIOR;} \\
\text{FORWARD ROUTINE complement ();} \\
\text{PORT a ENTRY (complement);} \\
\text{ROUTINE complement ();} \\
\text{b = (0) NOT a;} \\
\text{ENDROUTINE complement;} \\
\text{ENDBEHAVIOR;} \\
\text{ENDMODEL invert;}
\]

The syntax of the behavior model is explained in detail in the following sections.
2.2.1 Model Definition Header

The model definition header consists of the MODEL keyword, the model name, and the names and bit subscripts of the outputs and inputs (ports). It can also include parameter definitions for timing delays. See Section 2.9 for more information about defining parameters.

The keywords MODEL and ENDMODEL indicate to DECSIM that the enclosed information is a model that can be instantiated.

The model_name can contain from 1 to 127 characters. See Section 2.1.3 for more information on names.

The ports (signal connections) connect to something external to the model (often another model). Ports are specified with the port name and its bit subscripts. See Section 2.1.6 for more information on bit subscripts.

Example

```
MODEL 7400 [p1=2, p2=3NS, p3] out<31:0> = in1<31:0>, in2<31:0>;
```

If there are delay parameters, they are defined within square brackets () after the model name. Output signals are on the left side of the equal sign (=); input signals are on the right side. Bidirectional ports can be on either side of the equal sign.

Noninterconnected models have no connection (port) list, just an equal sign (=).

Example

```
MODEL model_without_ports = ;
```

2.2.2 BEHAVIOR and ENDBEHAVIOR Keywords

The keywords BEHAVIOR and ENDBEHAVIOR indicate that the enclosed model is behavior (not structure).

Directives and modifiers can be placed after the BEHAVIOR keyword. Valid modifiers for the BEHAVIOR header are TWOSTATE, FOURSTATE, CONVxx, and STATIC. See Section 2.5 for more information on modifiers. See Section 2.8 for more information on directives.

2.3 Module Definition

A module is a fragment of a model, contained in a separate file, that is compiled prior to the model accessing it.

Modules allow you to separate a large behavior model into smaller parts. Benefits include easily dividing a project among engineers, and reducing recompile time for changes that do not affect all the modules. When you correct an error, you have to recompile only the changed module and the model accessing it. There is a limit of 63 modules per model.

You can have only one module per file. The name of the file must be the same as the name of the module.
The module syntax closely resembles behavior model syntax.

**Syntax**

```plaintext
MODULE module_name;
  BEHAVIOR [subblock_name] [modifiers] [[directive[, ...]]];
  [declaration]...
ENDBEHAVIOR [subblock_name];
ENDMODULE module_name;
```

The example on the next page has the same functionality as the one used to show model definition in Section 2.2.

**Example**

```plaintext
!file change.bds
MODEL change b<0> = a<0>;  !model containing the module
BEHAVIOR {MODULES = (changelog)};  !MODULE directive
EXTERNAL ROUTINE complement();  !declaring routine in module
PORT a ENTRY (complement);  !referencing routine in module
ENDBEHAVIOR;
ENDMODEL change;

!file changem.bds
MODULE changem;
  BEHAVIOR;
  PORT a<0> INPUT, b<0> OUTPUT;  !modules must declare bit subscripts of
  ROUTINE complement ();  !ports and declare ports INPUT or OUTPUT
  b = 0 \ NOT a;
ENDROUTINE complement;
ENDBEHAVIOR;
ENDMODULE changem;
```

**2.3.1 How a Module Differs from a Behavior Model**

- Modules must be referred to in a MODULES directive in the BEHAVIOR header in the model. See Section 2.8.3 for more information on the MODULES directive.

- As with a behavior model, you can have only one module per file. However, the module name must be the same as the file name. The first character must be a letter, and the rest of the characters must be letters or numbers. The module name must be no more than 21 characters.

- The module header is like the model header except that it has no connection (port) list and no equal sign.

- Because there are no ports in the module header, modules that reference ports must declare each port with a PORT declaration. You must also specify the bit subscripts of the port, and whether it is an input or output. See the following sections for more information on modifiers for modules.
Behavior Primitives

- Those ports and global states declared in the model and used in the module must also be declared in the module.
- The same declarations apply to modules and models, except that a REVISION declaration in a module is ignored.
- The same modifiers apply to modules and models, except that modules cannot contain the ENTRY, CONVxx, or INITIAL modifiers.
- The MODULES directive cannot be used in a module.

2.3.2 PORT in Modules

Ports that are referenced in modules must be explicitly declared with the PORT declaration.

The port declaration must contain the name and width of the port, as well as an INPUT or OUTPUT modifier. The name and the subscripts of the module port declaration must be identical to the name and subscripts used in the model header.

2.3.2.1 INPUT

The INPUT modifier must be used with the PORT declaration in modules to declare an input port.

Syntax

```
INPUT
```

Example

```
PORT a<3:0> INPUT;
```

2.3.2.2 OUTPUT

The OUTPUT modifier must be used with the PORT declaration in modules to declare an output port.

Syntax

```
OUTPUT
```

Example

```
PORT b<7:0> OUTPUT;
```
2.3.2.3 Compiling Models and Modules

To compile a behavior model, you use the COMPIL/BEHAVIOR command (or COMBEH), which compiles the behavior model source file (default extension .BDS) and creates a file with the extension .NLB. You then use the COMPIL command, which compiles the top-level model (network description file - default extension .NET) and creates files with the extensions .NOB and .EXE.

To compile a module, you use the COMPIL/BEHAVIOR command (or COMBEH), which compiles the module source file and creates a file with the extension .SLB. Then you use COMPIL/BEHAVIOR (or COMBEH) on the module source files that access the module. Finally, you compile the network description file (default extension .NET) using the COMPIL command, which creates files with the extensions .NOB and .EXE.

For example, to compile the module shown in Section 2.3, you follow this sequence of commands:

1  RUN DECSIM$SYS:DECSIM
2  COMBEH changem.bds !compiling the module source file
3  COMBEH change.bds !compiling the model source file
4  COMPIL change.net !compiling the model network file

By dividing a large model into modules, you can save time in recompiling after making changes to some of the modules. You have to use COMBEH only on the module you changed and any models that access it. The other modules don’t have to be recompiled with COMBEH.

Note: Each module can be accessed by only one model in a network.

2.4 Declarations

Declarations define (declare) model or module components and characteristics.

All ports, states, synonyms, constants, and routines must be declared before they are used. The following sections describe the declarations and their modifiers.

Declarations can be either local or global. Local declarations are contained in a routine, and the declared object is available only to that routine. Global declarations are not in a routine, and make the declared object available to the entire model.
4.1 PORT

Describes an input or an output to an interconnectable behavior model. Outputs are always fourstate; inputs default to the model default, which is twostate. Ports are always global.

Port names and bit subscripts must be declared in the MODEL header. PORT declarations are required only for using PORT modifiers. You can specify only one entry routine per port.

Modifiers give DECSIM additional information on how declarations such as PORT work. Valid modifiers for PORT are BIDIRECTIONAL, TWOSTATE (input PORTS only), FOURSTATE, ENTRY, CONVxx, UNCONNECTED_OK, INPUT (for modules), and OUTPUT (for modules). See Section 2.5 for more information on modifiers.

Syntax

PORT name [ < bit_subscript> ] [ modifier... ] [ ,...]

Example

MODEL change b<0> = a<0>; !port names and width in model header
BEHAVIOR;
FORWARD ROUTINE complement ()
PORT a CONVU1 CONVZ0 ENTRY (complement); !PORT declaration with !modifiers
ROUTINE complement ()
b = (0) NOT a;
ENDROUTINE complement;
ENDBEHAVIOR;
ENDMODEL change;

2.4.1.1 Assigning Values to Ports

In the .BDS (model source) file, assignments can be made to output and bidirectional ports only. Assignments to these ports must be delayed, even when the delay is zero.

Example

PORTA = (0) 1;

Multiple delayed assignments can be made to behavior outputs. You can schedule a delayed assignment for any time that is greater than or equal to the time of the last delayed assignment for that variable. Once you have scheduled an event for a variable, you cannot retract it.
Examples

\[
\begin{align*}
\text{out1} &= \{10\}1; & \text{!out1} &= 1 \text{ at 10NS} \\
\text{out1} &= \{20\}0; & \text{!out1} &= 0 \text{ at 20NS} \\
\text{out2} &= \{20\}0; & \text{!out2} &\rightarrow 0 \rightarrow 1 \text{ at 20NS} \\
\text{out3} &= \{20\}1; & \text{!out3} &= 1 \text{ at 20NS} \\
\text{out3} &= \{10\}0; & \text{!results in a warning}
\end{align*}
\]

The problem illustrated in the second example above can be avoided by writing a routine that contains logic to determine whether or not you changed your mind. Activate this routine for the exact time of the desired event. In the example below, routine assign_to_X determines that this assignment is an additional assignment to output X. The new value, 123, is then assigned to X.

Example

\[
\begin{align*}
\text{STATE} \\
\text{assignment_sequence} &<31:0>, \\
\text{last_assign} &<31:0>; \\
\text{ROUTINE assign_to_X (val<31:0>, seq<31:0>);} \\
\text{IF seq GTR last_assign THEN} \\
\text{BEGIN} \\
\text{X} &= \{0\}\text{val} \\
\text{last_assign} &= \text{seq}; \\
\text{END;} \\
\text{ENDROUTINE assign_to_X;} \\
\ldots \\
\text{assignment_sequence} &= \text{assignment_sequence} + 1;
\end{align*}
\]

The routine would be activated by the following command:

\[
\text{SIM> activate (10) assign_to_X (123, assignment_sequence)}
\]

From the interactive command language, assignments to input ports must be made using the PATTERN command. However, for 1-bit ports, you can use DEPOSIT to assign a value to a port at the %NET (network) level. Values cannot be deposited or patterned to output or bidirectional ports.

2.4.1.2 Reading from Output Ports

%read_output_port is a system variable that allows you to use the value of an output port in an expression. It also allows you to print the value of an output port. See Section 2.6.14.
Behavior Primitives

Example

MODEL add c2<0>, sum<1:0> = a<0>, b<0>, c<0>; BEHAVIOR;
FORWARD ROUTINE add1;
PORT
  a ENTRY (add1),
  b ENTRY (add1),
  c ENTRY (add1);
ROUTINE add1();
  sum = (0) a + (width = 3) b + (width = 3) c;
  c2 = (0) ^read_output_port (sum<1>); !value of most significant
      !bit is assigned to c2
WAIT (10);
  ^print ('c2 = ', ^read_output_port (c2), 'sum = ',
           ^read_output_port (sum));
ENDROUTINE add1;
ENDBEHAVIOR;
ENDMODEL add;

2.4.2 STATE

Defines a variable and allocates storage. You must specify a width,
included by bit subscripts, for the state. The maximum width is 256
bits.

If you specify a bit subscript with no values (<>, the width defaults to the
width and bit subscripts of the host machine. For example, the default on
a VAX is 32 bits and <31:0>.

Syntax

STATE name [word_subscripts] bit_subscripts [modifier] [ , . . . ]

Valid modifiers for STATE are TWOSTATE, FOURSTATE, ENTRY (for
global states), CONVxx (for global and local states), STATIC (for local
states), and INITIAL (for global states and static local states). See
Section 2.5 for more information on modifiers.

Example

STATE temporary <>; ! 32 bits
  STATE memory [0:1000]<7:0>,
  register<31:0> STATIC,
  csr<12:0>, ! 13 bits
  flag<0>; ! 1 bit

By specifying a word subscript range for a state, you can represent an
array. Arrays cannot be fourstate. The following example declares a 4096
word by 16 bit array.

Example

STATE m[0:4095]<15:0>;

Note: Models execute much faster if they use 32-bit twostate states
whenever possible.

2–14
States can be local or global.

A local state is declared inside a routine, and cannot be referenced or modified by another routine. Assignments to local states must be direct, not delayed.

Global states are declared outside a routine. The contents of global states are retained throughout simulation; any routine can reference or modify them. Assignments to global states can be either delayed or direct, except for assignments to arrays, which must be direct.

Note: You can make a delayed assignment to an array by first using the SYNONYM declaration to give the word an alternate name. For example, by declaring SYNONYM word4<same_bit_size> = arr[4]; you can then make a delayed assignment to word4.

Example

```aldebro
MODEL assign_model = ; !stand-alone model -- no ports
BEHAVIOR;
STATE sum1<7:0>, !global state
     sum2<7:0>; !global state
ROUTINE assign ();
STATE a<3:0>, !local state
     b<3:0>; !local state
     sum1 = a + b; !direct assignment
     sum2 ={5} a + b; !delayed assignment
ENDROUTINE assign;
ENDBEHAVIOR;
ENDMODEL assign_model;
```

Local states store data in either of two ways: with dynamic or static storage.

The default is dynamic local states. They only exist when the routine containing them is executing. Dynamic states are initialized to 0 when the routine containing them is called or activated.

A static local state exists from one routine activation to another, and is not reinitialized to 0 every time the routine is invoked. Like any local state, it cannot be referenced or modified from another routine. You specify a static local state with the STATIC modifier.

Behavior states can be used to trigger WATCH or TRACE commands.

2.4.2.1 Communicating with States

Each invocation of a routine has its own copy of that routine's dynamic local states. This copy cannot be shared with other invocations of that routine. However, one invocation of a routine can communicate with another invocation through static local states or global states. Routines can communicate with other routines through global states.

Each instantiation of a model has its own unique states, that cannot be shared with other instantiations of that model. However, models can communicate with other models through ports.
2.4.2.2 Assigning Values to States
In the .BDS (source) file, to assign a value to a global state, use either delayed or direct assignment.

Example

    globalstatea = (0) 1
    globalstatea = 1;

In the .BDS (source) file, to assign a value to a local state, use direct assignment.

Example

    localstateb = 1;

To assign a value to a global state or static local state from the interactive command language, use the EVALUATE command.

Example

    EVALUATE yourstate = 11

You can't assign a value to a dynamic local state from the interactive command language until the routine containing it has been invoked, because the state exists only for the duration of the routine invocation. You need a combination of interactive command language WATCH and BREAK or EVALUATE commands to interrupt the routine.

2.4.3 SYNONYM

Creates an alternate name for a previously defined state or port. It is commonly used to create a simple name for a complex, frequently used field. The alternate name appears to the left of the equal sign; the previously defined state or port appears to the right. Synonyms are accessible from the DECsim command language.

You can apply SYNONYM to global states by using it outside a routine. You can apply SYNONYM to local states by using it inside the routine containing the local state.

You cannot concatenate with the SYNONYM declaration; concatenation is used in an expression to dynamically construct a value, whereas SYNONYM creates only an alternate name.

You cannot declare a synonym for another synonym.

Syntax

    SYNONYM new_name [ [word_sub] ] <bitsub>
    = existing_name [ [word_sub] ] [ <bitsub> ] [ , . . . ] ;
Example

```
STATE word<15:0>;
SYNONYM byte0<7:0> = word<7:0>,
    byte1<7:0> = word<15:8>;
SYNONYM pc<31:0> = gpr[15]<31:0>,
    ac[0:7]<0:35> = m[0:7]<0:35>;
```

### 2.4.4 CONSTANT

Declares a name that represents a single unchanging value. With the CONSTANT declaration, you declare a name and the literal value to be bound to that name.

Constants can be expressed in radix 2 through radix 16. The default is radix 10.

Constants must be 32 bits or less if the model is instantiated more than once in the network. Constants must be twoostate.

**Syntax**

```
CONSTANT name = number [, ...]
```

where number can be either a literal or the name of a previously declared constant

See Section 2.1.8 for more information on numbers.

**Example**

```
CONSTANT A=1, B=2, C=3;
CONSTANT m1 = 3FF80000$16, m2 = m1;
```

Signs (+ or −) cannot be used with the literal in a CONSTANT declaration.

If you use a constant in a WAIT or ACTIVATE statement or in delayed assignments, it must be enclosed in parentheses. For Example:

```
CONSTANT int = 128;

ROUTINE generate;

WAIT(int);
```

See Section 2.8.4.5 for more information on the syntax for time.
2.4.5 REVISION

Indicates the revision number of the behavior model. There can be only one REVISION declaration in a model. REVISION is a global declaration, and cannot be placed in a routine or a module.

Syntax

REVISION number

Example

REVISION 17; !Edited on 2-Dec-84

During your simulation, you can display the revision number with the LABEL or EXAMINE/LABEL command.

2.4.6 REQUIRE

Places the contents of another source language file into the behavior model.

When the compiler processes a REQUIRE declaration, it suspends reading the original file and starts reading the file specified with the REQUIRE declaration. When the compiler finishes reading that "required" file, it resumes reading the original file at the point following the REQUIRE declaration.

A REQUIRE file can contain another REQUIRE declaration. The maximum nesting varies according to the host system open-file limit and the number of library files referenced, but it is always at least three.

REQUIRE is commonly used to specify files containing source code, such as declarations that are common to several model or module source files.

REQUIRE cannot be inserted in a statement.

Syntax

REQUIRE file_name [ , ... ]

Example

REQUIRE [randall.models]decls;
REQUIRE ' [randall.models]decls.nrq:4 ';

The default file extension is .NRQ.

The file name must be enclosed in single quotation marks (' ') if it contains a semicolon.
2.4.7 INFORM

Sends a message to the terminal during compilation when the INFORM statement is parsed. By strategically placing INFORM statements in your model, you can follow the progress of the compilation.

See Section 2.1.5 for more information on string syntax.

Syntax

```
INFORM string
```

Example

```
INFORM 'macro definitions have been parsed';
INFORM 'Model being compiled with LRU cache selected';
```

The message is limited to 132 characters.

2.4.8 MACRO

Declares or defines a macro that is later expanded to text. A macro is expanded at compile time, rather than at run time. Macros defined in the behavior model cannot be accessed from the interactive command language. However, macros are expanded in the listing (.LIS) file, which is created with COMBEH source_file/LIS.

2.4.8.1 Defining Macros

Macros are defined in the behavior model. A macro definition is always global, whether it is defined inside a routine or not.

Formal parameters are placeholders or dummies that are enclosed in brackets ([]) following the macro name. You can specify a default value for each formal parameter. It is used when you call the macro without supplying actual parameters. Default values must be enclosed in single quotation marks (').

The following restrictions are in force for macro definitions:

- Multiple statements should be enclosed in BEGIN–END blocks to avoid potential problems when the macro is expanded.
- Do not put comments in a macro definition; they can cause problems when the macro is expanded.
- The last line of the macro definition (just before $ENDMACRO) should not end with a semicolon (;). Otherwise, when the macro is called (followed by a semicolon), there will be two consecutive semicolons.
- The keywords MACRO, ENDMACRO, AND $ENDMACRO should not be used inside the macro body, not even within the text of a %PRINT statement.
Behavior Primitives

Syntax

MACRO name [[formal_parameter_name [= default,...][,...] ]] =
macro_body $ENDMACRO

Example

MODEL muchro = ;
BEHAVIOR;
   STATE PC<31:0>;
   MACRO error [''no error message'' ] =
      BEGIN
         %PRINT ('Error: ',errmsg);
         %PRINT ('pc = ', pc);
         HALT;
      END
   $ENDMACRO;
   ROUTINE make ();
      error [''divide by 0'' ];
   ENDROUTINE make;
   ROUTINE make1 ();
      error [''high bits truncated'' ];
   ENDROUTINE make1;
   ROUTINE make2 ();
      error;
   ENDROUTINE make2;
ENDBEHAVIOR;
ENDMODEL muchro;

The above example shows a model containing a macro definition. Note
the use of single quotation marks ('' ). Because a text string (which must
be enclosed in single quotation marks) is used as an actual parameter
(which must be enclosed in single quotation marks), it must be enclosed in
consecutive single quotation marks within single quotation marks.

The following example shows the result of calling the three routines.

Example

SIM> CALL MAKE
Error: divide by 0
pc = 00000000
SIM> CALL MAKE1
Error: high bits truncated
pc = 00000000
SIM> CALL MAKE2
Error: no error message
pc = 00000000

In a macro body, the dollar sign ($) is a text concatenation lexical operator.
It can combine formal parameters with other text. The dollar sign does
not appear in the macro expansion. The following example concatenates
STATE_ with the formal parameter p1.
Example

MACRO xx [pa1] =
STATE $pa1 = 0   !macro definition
$ENDMACRO;
STATE state_01<0>, state_02<0>, state_aa<0>;    !states declared
xx ['01'];
xx ['02'];
xx ['aa'];
STATE_01<0> = 0    !result of macro call
STATE_02<0> = 0
STATE_AA<0> = 0

2.4.8.2 Using Macros

You call a macro by using the macro name and any actual parameters. You can pass parameters to the macro by placing the actual parameters in brackets ([]). Each actual parameter must be enclosed in single quotation marks (' '). If an actual parameter is omitted, and there is no default parameter, the empty string is used.

When passing parameters, you have the option of using a name-based list or an order-based list. In an order-based list, the actual parameters are mapped to the formal parameters occupying the corresponding locations.

Syntax

macro_name [[[string[,...]]]]

Example

MACRO short [pa1,pa2] =
macro body    !macro definition
$ENDMACRO;
short ['sa','sb']; !macro call - the value 'sa' is substituted for pa1; the value 'sb' is substituted for pa2.

When using name-based parameter passing, the order of the actual parameters is not important. If you don’t specify a parameter, the default value is used.

Syntax

macro_name [[[parameter_name = string[,...]]]]
Behavior Primitives

Example

MACRO short [pa1,pa2] = !macro definition
    macro body
SENDMACRO;
short [pa2='sb',pa1='sa']; !calling the macro

The following example shows a macro and a listing (/LIS) file, which shows how the macro is expanded when called.

Example

MODEL test = ;
BEHAVIOR;
    FORWARD ROUTINE sam();
    ROUTINE sam();
    macro hi [message = '''default text'''] =
        $print (message)
        $endmacro;
    hi;
    hi ['''actual text, 1st macro call'''];
    hi ['''actual text, 2nd macro call'''];
ENDROUTINE sam;
ENDBEHAVIOR;
ENDMODEL test;

The following listing file (created with COMBEH filename/LIS) illustrates how macros are expanded just prior to the macro call. The macro definition is three lines, so three lines are inserted just prior to each macro call.

! SX version V4.7-2456 Compilation on 22-NOV-1985 at 09:34:18 Listing page 1
!
! Source file "(LVSS:[RANDB.BK MODELS MOSTLYBEHAVIOR]MACRO.BDS;9)
LVSS:[RANDB.BK MODELS MOSTLYBEHAVIOR]MACRO.BDS;/LIB/LI:
  0    1 MODEL test = ;
  2    2 BEHAVIOR;
  3    3     FORWARD ROUTINE sam();
  4    4     ROUTINE sam();
  5    5     macro hi [message = '''default text'''] =
  6    6     $print (message)
  7    7     $endmacro;
  8    8     hi;
  9    9     $print (''default text'')
 10   10     hi;
 11   11     $print (''actual text, 1st macro call'')
 12   12     hi [''actual text, 1st macro call''];
 13   13     $print (''actual text, 2nd macro call'')
 14   14     hi [''actual text, 2nd macro call''];
 15   15     ENDROUTINE sam;
 16   16     ENDBEHAVIOR;
 17   17     ENDMODEL test;
2.4.9 ROUTINE

Defines the heart of a behavior model. A ROUTINE declaration consists of a ROUTINE header, declarations and statements, and the ENDROUTINE keyword. All operations during simulation, such as assignments, addition, and shifting, are performed by statements in a routine.

This section defines a routine. Section 2.6.17 describes calling a routine, and Section 2.7.7 describes using a routine call in an expression.

Syntax

ROUTINE routine_name [<bit_subscript>] [modifiers] [directives] [([formal_parameters])];
    [ declarations; ] ...
    [ statements; ] ...
ENDROUTINE routine_name;

Example

MODEL change b<0> = a<0>;
BEHAVIOR;
    FORWARD ROUTINE complement ();
    FORWARD ROUTINE display ();
    PORT a ENTRY (complement);  !specifies a routine to be
                                !activated when 'a' changes
    ROUTINE complement ();
        b = (0) NOT a;
        display();  !routine call
    ENDROUTINE complement;
    ROUTINE display ();
        %PRINT ('a is ', a);
    ENDROUTINE display;
ENDBEHAVIOR;
ENDMODEL change;

2.4.9.1 Routine Header

The ROUTINE header consists of the ROUTINE keyword, routine_name, bit subscripts of the return value (if there is one), modifiers, directives, and formal parameters. Valid modifiers for the routine header are TWOSTATE, FOURSTATE, STATIC, and CONVxx. See Section 2.5 for more information on modifiers.

Syntax

ROUTINE routine_name [<bit_subscript>] [modifiers] [directives] [([formal_parameters])]

The ROUTINE keyword indicates the ROUTINE header.

The routine_name follows standard naming rules. See Section 2.1.3 for more information on naming rules. The routine_name should not have the same name as another global object, such as a port, state, constant, or synonym.
Behavior Primitives

Examples

ROUTE x;
ROUTE x ();
ROUTE return_value <31:0> ();
ROUTE two_arguments (arg1<> ,arg2<>);
ROUTE give_and_take <31:0> (arg1<> ,arg2<>);

The following restrictions are in force for routine declarations:

- All declarations in a routine are local to that routine.
- Routines cannot be nested; however, they can invoke other routines.
- Recursive routines do work; however, DECSIM does not support them, and recommends that they be avoided.
- The number and complexity of statements in routines is limited by the BLISS–32 compiler. Thus, routines should be limited to approximately 300 statements.

2.4.9.2 Return Value

A routine returns a value only if a RETURN statement supplies that value. The width of the return value in the routine must be declared by using bit subscripts in the routine header.

The bit_subscripts are enclosed in angle brackets. If the routine does not return a value, omit the bit_subscript and angle brackets in the routine header.

Routines that return a value must be called, not activated. See Section 2.6.11 for more information on return values.

2.4.9.3 Formal Parameters

Formal parameters are local variables that are given initial values from the actual parameters specified when the routine is invoked. Formal parameters include the name and bit subscripts, and are enclosed in parentheses in the routine header. If there are no formal parameters, you can omit the parentheses in the routine definition.

Syntax

formal_parameter_name <bit_subscripts> [modifiers] , ...

Example

ROUTE f<31:0>(x<31:0> TWOSTATE, y<31:0>);

In the following examples, the command language passes two values into a routine, which returns the 32-bit product of the two 32-bit arguments (x and y).
Behavior Primitives

Example

MODEL pass = ;
BEHAVIOR;
ROUTINE mul<31:0> (x<31:0>, y<31:0>);
  RETURN (x*y);
ENDROUTINE mul;
ENDBEHAVIOR;
ENDMODEL pass;

The following example illustrates calling the routine from the interactive command language.

Example

SIM> CALL mul (5, 9)
SIM> EVALUATE %ROUTINEVALUE
  45
SIM> CALL mul (15, 15)
SIM> EVALUATE %ROUTINEVALUE
  225

The valid modifiers for formal parameters are CONVxx, FOURSTATE, and TWOSTATE. These modifiers are described in Section 2.5. Other modifiers are ignored. The default modifier for formal parameters is TWOSTATE.

In the following examples the command language passes a FOURSTATE value into a routine that prints the value.

Example

MODEL form_prm = ;
BEHAVIOR;
ROUTINE display (prm1<0> FOURSTATE);
  %PRINT(prm1);
ENDROUTINE display;
ENDBEHAVIOR;
ENDMODEL form_prm;

In the following example the routine is called from the command line and passed a 4-state value.

Example

SIM> CALL display(U#2)
U#2

2.4.9.4 Actual Parameters

Actual parameters are expressions in a routine invocation that are evaluated when the routine is invoked. Their values initialize the corresponding formal parameters in the routine. Actual parameters are enclosed in parentheses after the routine name, and are separated by commas. The maximum number of actual parameters that may be passed to a routine is 256. See Section 2.6.17 for more information on routine calls.
Behavior Primitives

Example

mul (6, 8);  \[ \text{routine call where 6 and 8 are actual parameters} \]
ACTIVATE no_return_value (7, 2);  \[ \text{routine activation where 7 and 2 are actual parameters} \]

2.4.9.5 Modifiers

Modifiers in the routine header affect the routine’s return value and create a new default, which can be overridden, for all parameters and states declared in the routine. See Section 2.5 for more information on modifiers.

2.4.9.6 Directives

Directives in the routine header create a new default, which can be overridden, for statements and operations in the routine.

2.4.10 FORWARD ROUTINE

FORWARD ROUTINE is a global declaration that allows a model to reference a routine before it is defined. FORWARD ROUTINE is normally used before a PORT declaration with an ENTRY modifier. If you use FORWARD ROUTINE declarations, you don’t have to order the routines.

The FORWARD ROUTINE declaration consists of the keyword FORWARD and an exact copy of the routine header. See Section 2.4.9.1 for information on the routine header.

Syntax

FORWARD routine_header

Example

FORWARD ROUTINE two_arguments (arg1<> , arg2<> );
FORWARD ROUTINE f<31:0>(x<31:0>);
FORWARD ROUTINE do_something();

2.4.11 INITIALIZE ROUTINE

The INITIALIZE ROUTINE declaration is exactly like the FORWARD ROUTINE declaration except that the routine is scheduled for activation in the first time slot of the simulation (time = 0ns). The routine is executed immediately after the SIMULATE command is given.

If a routine has formal parameters, when it is invoked during the first time slot of the simulation it uses 0 as the values for the parameters. After the first time slot it uses the values of the actual parameters passed to it by another routine or from the command line.

Only one routine per model can be the INITIALIZED routine, but this routine can call other routines.

The INITIALIZE ROUTINE declaration consists of the keyword INITIALIZE and an exact copy of the routine header. See Section 2.4.9.1 for information on the routine header.
Behavior Primitives

Syntax

INITIALIZE routine_header

Example

INITIALIZE ROUTINE called_at_ons();

2.4.12 EXTERNAL ROUTINE

EXTERNAL ROUTINE is a global declaration that allows a routine in a model to invoke a routine declared in a module. It allows a routine in a module to invoke a routine declared in another module or in the model.

The EXTERNAL ROUTINE declaration consists of the keyword EXTERNAL, and an exact copy of the routine header. See Section 2.4.9.1 for information on the routine header.

Syntax

EXTERNAL routine_header

Example

EXTERNAL ROUTINE two_arguments (arg1<> , arg2<> );

2.4.13 INCLUDE

INCLUDE is a global declaration used to call a routine written in a language other than the behavior language. It is always used with an EXTERNAL ROUTINE declaration that has the INCLUDED modifier (See Section 2.4.12 and Section 2.5.10).

The INCLUDE declaration provides information for the VMS linker when the model is compiled and linked. The INCLUDE declaration consists of the keyword INCLUDE followed by one of the following:

- A file specification for an .OBJ file. The .OBJ file is the output from the compiler or assembler for the other language. This file contains the compiled or assembled routine that your model calls.

- The file specification parameter and qualifiers for the LINK command enclosed in single quotes. Use this when the routine you want to call is in a library of compiled or assembled routines called an .OLB file. Specifying .OLB files requires the /LIB LINK command qualifier. For more information about the LINK command and the other qualifiers see the VAX/VMS Linker Reference Manual.
Behavior Primitives

Syntax

INCLUDE obj_file_specification;
INCLUDE 'LINK_command_file_specification_parameter/qualifiers';

Example

INCLUDE [user]c_routine.OBJ;
INCLUDE '[user]c_routines.OLB/LIB';

2.5 Modifiers

Modifiers give DECSIM additional information on how declarations work. They can be specified for individual states and ports, as well as for models and routines. Modifiers in the behavior header create new defaults, which can be overridden, for the entire model. Modifiers in the routine header create new defaults, which can be overridden, for that routine. Multiple modifiers can be used. Table Table 2–1 shows where the various modifiers can be used.

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Global STATE</th>
<th>Local STATE</th>
<th>PORT</th>
<th>ROUTINE Header</th>
<th>BEHAVIOR Header</th>
<th>EXTERNAL ROUTINE</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT</td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIDIRECTIONAL</td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TWOSTATE</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td>inputs only</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>FOURSTATE</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>[NO]ENTRY</td>
<td>yes*</td>
<td></td>
<td></td>
<td>input or bidirectional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONVxx</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td>yes</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>UNCONNECTED_OK</td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STATIC</td>
<td>yes</td>
<td></td>
<td></td>
<td>static only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INITIAL</td>
<td>yes</td>
<td>static</td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*only effective with delayed assignments in event-driven simulation

The INPUT and OUTPUT modifiers are sometimes used in PORT declarations in large models for documentation purposes; their primary use, however, is in PORT declarations in modules and they are described in Section 2.3.2. The other modifiers are described in the following sections.
2.5.1  **BIDIRECTIONAL**

Specifies that the port is both an input and an output. Bidirectional ports are four-state.

Values can be assigned to bidirectional ports within the behavioral model, as they can be assigned to other ports. However, you cannot use the Command Language DEPOSIT or PATTERN commands to assign values to them.

One way to assign values to bidirectional ports using the Command Language is to create a structural shell around the behavioral model. You can then DEPOSIT or PATTERN to the inputs of the shell. Another method is to include dummy global states and a routine that assigns the global states to the bidirectional ports. Then, from the Command Language, you can use the EVALUATE command to assign values to the states and the ACTIVATE command to activate the routine (see the example of driving a behavioral bidirectional port in the **DECSIM User's Guide**).

**Syntax**

```
BIDIRECTIONAL
```

**Example**

```
MODEL tristate_bus io_bus<31:0> = decode1<0>, decode2<0>, decode3<0>;
PORT io_bus BIDIRECTIONAL,
    decode1 BIDIRECTIONAL;
```

2.5.2  **TWOSTATE**

Specifies that 0 (zero) and 1 (one) are the only possible values of a bit. TWOSTATE is the default. It cannot be specified for output ports.

**Syntax**

```
TWOSTATE
```

**Example**

```
PORT in_1 TWOSTATE;
PORT in_2 TWOSTATE CONVU0 CONVZ1;
STATE example<3:0> TWOSTATE;
ROUTINE generate<31:0> TWOSTATE (arg1<31:0>, arg2<31:0>);
BEHAVIOR TWOSTATE;
```
Behavior Primitives

By default, behavior models are twostate. Inside the model, DECSIM converts both U's and Z's to zero (0). However, outside the model, at the top (network) level, all signals are fourstate.

You can override the default conversion of U and Z with the CONVxx modifiers. CONVxx modifiers do not work on twostate states. See Section 2.5.6 for information on the CONVxx modifiers that can be used with TWOSTATE.

### 2.5.3 FOURSTATE

Specifies that 0 (zero), 1 (one), U#2 (undefined), and Z#2 (high impedance) are the four possible values of a bit.

**Syntax**

```
FOURSTATE
```

**Example**

```
PORT sample FOURSTATE;
STATE temp1<3:0> FOURSTATE;
ROUTINE mul<7:0> FOURSTATE (arg1<7:0>,arg2<7:0>);
BEHAVIOR FOURSTATE;
```

FOURSTATE applies to all operators (see Appendix D), but works only with unsigned arithmetic.

### 2.5.4 ENTRY

Specifies a routine to be activated when an input port or state value changes. The state value must change as the result of a delayed assignment.

The specified routine cannot have formal parameters in its header. Such routines are passed values for the parameters when they are activated and a transition in a port or state provides no means of passing specified values to these routines.

A routine may be entered on all transitions of the named signal. For example, PORT in1 ENTRY (swap) will activate routine swap for any change in in1, including U_0. You can also specify specific transitions. For example, PORT in1 ENTRY1_0 (swap) will activate the routine only if in1 changes from 1 to 0.

**Note:** At the top (%NET) level, outside the Behavior model, all signals are fourstate. Inside the Behavior model, inputs are twostate by default. U#2 and Z#2 are converted to 0 by default or to 1 if a CONVU1 or CONVZ1 modifier is used. In DECSIM Version 5.2 or higher, the ENTRY modifier sees the value before the CONVxx modifier. A CONVU0 modifier, for example, translates an initially
undefined port to 0. If you then assign a 0 to that port, the routine will be activated.

There is an ENTRY state for all combinations of legal signal values (0, 1, U, Z). The "don't care" value of X is also valid.

ENTRY state overrides an additional ENTRY modifier for that transition. For example,

\[
\text{PORT clk ENTRY (rtn_a) ENTRY1_0 (rtn_b)}
\]

calls rtn_a for all transitions except when clk changes from 1 to 0, in which case it calls rtn_b.

The routine must be declared (with FORWARD ROUTINE, EXTERNAL ROUTINE, or the ROUTINE declaration) before the PORT declaration. ENTRY is not valid for local states or for output ports.

The ENTRY modifier can reference only one routine; however, that routine can invoke other routines.

Several ports can specify the same entry routine. However, if input changes are simultaneous, the routine is called one time only. If you want the routine to run several times, specify a different routine for each port, with each of these routines calling a common routine.

**Syntax**

\[
\text{ENTRY} \text{[state state]} (\text{routine name})
\]

**Example**

\[
\begin{align*}
\text{PORT in1 ENTRY (rtn_a);} & \quad \text{!activate routine on any transition} \\
\text{PORT in1 ENTRYU_1 (rtn_a);} & \quad \text{!activate routine when signal changes from 0 to 1} \\
\text{PORT in1 ENTRY0_X (rtn_a);} & \quad \text{!activate routine if signal changes from 0 to anything} \\
\text{PORT in1 ENTRY (rtn_a) ENTRY1_0 (rtn_b);} & \quad \text{!activate rtn_b if signal changes from 1 to 0;} \\
& \quad \text{!otherwise, activate rtn_a} \\
\text{STATE C<> ENTRY (rtn_a);}
\end{align*}
\]

**2.5.5 NOENTRY**

Specifies that no routine will be invoked. This is the default.
Behavior Primitives

Syntax

NOENTRY

Example

PORT in2 NOENTRY;
STATE reg<7:0> NOENTRY;

2.5.6 CONVxx

Specifies the translation of undefined (U#2) and high-impedance (Z#2) values into values 0 and 1, which can then be manipulated in a model.

The CONVxx modifiers are:

- CONVU0 – converts U to 0 – this is the default
- CONVU1 – converts U to 1
- CONVZ0 – converts Z to 0 – this is the default
- CONVZ1 – converts Z to 1

CONVxx modifiers can be used with two-state and four-state ports and with four-state states. They can also be used with four-state data to declare the conversion pattern in a four-state to two-state assignment statement. CONVxx modifiers can be specified on the BEHAVIOR, STATE, and PORT declarations, and on the ROUTINE header.

Syntax

CONVU0

CONVU1

CONVZ0

or

CONVZ1

The following example illustrates the hierarchy of the different places where the CONVxx modifiers can be used.

Example

MODEL conv4to2 p4<3:0> = x4<3:0>, y4<3:0>, z4<3:0>;
!
! Convert modifiers on the BEHAVIOR statements set conversion
! patterns for all four-state PORTs and STATEs (global & local).
!
BEHAVIOR CONVZ1;
Behavior Primitives

! Convert modifiers on ROUTINE headers only affect local states
! declared in the routine.
!
FORWARD ROUTINE main CONVU1();
!
! Convert modifiers on fourstate PORTs and global STATES override
! any convert modifiers specified in the BEHAVIOR statement.
!
PORT
x4 FOURSTATE ENTRY (main),
y4 FOURSTATE CONVZ0 CONVU0;

STATE
a4<3:0> FOURSTATE,
b4<3:0> FOURSTATE CONVU1,
c4<3:0> FOURSTATE CONVZ0 CONVU1,
  a2<3:0>, b2<3:0>, c2<3:0>, d2<3:0>,
  e2<3:0>, f2<3:0>, x2<3:0>, y2<3:0>;

ROUTINE main CONVU1();
!
! Convert modifiers on LOCAL STATES override any conversion
! modifiers declared in the BEHAVIOR statement or ROUTINE header.
!
NOTE: If both convert modifiers are specified at the same level
(see "m4" below), the default CONVU0 will be used and
! CONVU1 ignored.
!
STATE
14<3:0> FOURSTATE,
m4<3:0> FOURSTATE CONVU0 CONVU1,
n4<3:0> FOURSTATE CONVZ0 CONVU0;

BEGIN
!
! Initial values on all 6 fourstate states will be
! 10ZU2 to illustrate conversion patterns.
!
a4 = 10ZU2;  b4 = 10ZU2;
c4 = 10ZU2;  i4 = 10ZU2;
m4 = 10ZU2;  n4 = 10ZU2;

a2 = a4;  ! a2 <- 1010  U changes to 0 by default.
!
! Z changes to 1 because of the modifier on the BEHAVIOR
! statement.

b2 = b4;  ! b2 <- 1011  U changes to 1 because of the
!
! modifier on the global STATE. Z changes to 1 because
! of the modifier on the BEHAVIOR statement.

! c2 = c4;  ! c2 <- 1001  U changes to 1 and Z to 0
! ! because of the modifiers on the global STATES.
! ! CONVZ0 overrides CONVU1 on the BEHAVIOR statement.

d2 = i4;  ! d2 <- 1011  U changes to 1 because of the
!
! modifier on the ROUTINE header. Z changes to 1
! because of the modifier on the BEHAVIOR statement.

! e2 = m4;  ! e2 <- 1010  U changes to 0 because CONVU0
! ! and CONVU1 are both specified at the same level.
! ! Z changes to 1 because of the modifier on the BEHAVIOR
! ! statement.

2-33
Behavior Primitives

\[
x_2 = x_4; \quad ! x_2 \leftarrow 0 \quad x_4 \text{ is } UUUU. U \text{ changes to 0 by default.}
\]

! Despite the conversion modifiers on the fourstate state/ports, an
! assignment from fourstate to fourstate does not do any conversions.
!
\[
p_4 = (0)a_4; \quad ! p_4 \leftarrow 10ZU \quad \text{no conversion since } p_4 \text{ is fourstate also}
\]

END;
ENDROUTINE main;
ENDBEHAVIOR;
ENDMODEL conv4to2;

Note: At the top (%NET) level, outside the Behavior model, all signals
are fourstate. Inside the Behavior model, inputs are twostate by
default. U\\#2 and Z\\#2 are converted to 0 by default or to 1 if a
CONVU1 or CONVZ1 modifier is used. In DECSIM Version 5.2 or
higher, the ENTRY modifier sees the value before the CONVxx
modifier. A CONVU0 modifier, for example, translates an initially
undefined port to 0. If you then assign a 0 to that port, the routine
will be activated.

You can use CONVxx modifiers in models, not modules. Use the CONVxx
or MAP operators to specify conversion patterns in modules.

2.5.7 UNCONNECTED_OK

Identifies input, output, or bidirectional ports to be left unconnected. This
modifier tells NETPRO that the port may be left out of the wirelist when
the model is instantiated. In the wirelist in the instance file, enter a null
signal (, ,) as the signal name you wish to leave unconnected. The signal
name should be omitted from the INPUT or OUTPUT declarations.

DECSIM generates a unique name for the unconnected port, in the form of
"$$n", where n is an integer. This name must be used to access the signal
with the Interactive Command Language.

Twostate ports initialize to 0 by default. Fourstate ports initialize to U by
default. Bidirectional ports initialize to U by default.

You can specify an initialization value on an input port to get a value
into the port. You cannot specify an initialization value on an output
port. For twostate ports with fourstate initialization values, U's and Z's
are converted to 0 by default. You can override the default by using the
CONVxx modifier. The result of the initialization is truncated if the width
of the initialization data exceeds that of the port. If the port is wider than
the initialization data, the result is zero-extended.
Example

```
PORT
in1<7:0> UNCONNECTED_OK,
in5<7:0> FOURSTATE UNCONNECTED_OK,
bi2<7:0> BIDIRECTIONAL UNCONNECTED_OK
in2<31:0> UNCONNECTED_OK (aafafafaf16),
in6<7:0> CONV2I CONVUI UNCONNECTED_OK
   (01ZUUI010#2),
in7<7:0> UNCONNECTED_OK (FFF16),
out1<31:0> UNCONNECTED_OK,
out2<7:0> UNCONNECTED_OK (ff16);
```

!default init to 0
!default init to U
!default init to U
!init value HEX, 32 bits:
!convert Z and U to 1
!initialize 8 bits with
!12, Will be truncated.
!results in a warning

---

2.5.8 STATIC

When the STATIC modifier is used in a local STATE declaration it specifies that there is a single copy of the local state that retains its value from one routine activation to another. It is not reinitialized to 0 every time the routine is invoked. Like any local state, it cannot be referenced or modified from another routine. STATIC applies only to local states.

When the STATIC modifier is used in the BEHAVIOR header it has the same effect as including the STATIC modifier in all the local STATE declarations in the subblock.

Syntax

```
STATIC
```

Example

```
STATE sum<31:0> STATIC;
BEHAVIOR STATIC;
```

---

2.5.9 INITIAL(value)

Specifies an initialization value for a global state or a static local state. The state must be two-state and 32 bits or less. It cannot be an array.

Syntax

```
INITIAL(initial_value[#radix_value])
```

Example

```
STATE a<7:0> INITIAL(FF16);
```
2.5.10 INCLUDED

The INCLUDED modifier specifies that a routine is not written in the behavior language. It is used only in an EXTERNAL ROUTINE declaration and only when an INCLUDE declaration is also used. The INCLUDED modifier specifies that the routine is not in the main module or any module in the model but is in the file specified in the INCLUDE declaration (see Section 2.4.13).

Syntax

INCLUDED

The following example contains an EXTERNAL ROUTINE declaration for a routine named generate with a formal parameter arg<31:0>.

The INCLUDED modifier specifies that the routine was not written in the behavior language and is not in any module in the model.

The INCLUDE declaration specifies that the file named 'c_routines.obj' contains the compiled 'generate' routine.

Example

MODEL foreign_routines b<7:0>=a<7:0>;
BEHAVIOR;
INCLUDE c_routines.obj;
EXTERNAL ROUTINE generate INCLUDED (arg<31:0>);

ROUTINE pass;
b=(10) generate(a);
ENDROUTINE pass;

ENDBEHAVIOR;
ENDMODEL foreign_routines;

2.6 Statements

Statements allow you to describe procedurally the behavior of the hardware being modeled. Statements are code that is executed during simulation when the routine in which it is contained is invoked. All statements appear within routines. Statements can be labeled (see Section 2.1.9). The following sections describe the behavior language statements.

2.6.1 BEGIN–END

Groups statements into blocks, creating a compound statement that is treated as a single statement. Statements in a block are executed sequentially. A RETURN or LEAVE statement interrupts this sequence.

Directives in the BEGIN–END block create a new default, which can be overridden, for statements and operations within the block.
Behavior Primitives

Syntax

BEGIN [directives] [statement;]...
END

Example

ROUTINE div <31:0> (a<31:0>, b<31:0>);
IF b EQL 0 THEN
BEGIN
  %PRINT ('Attempted division by 0');
  HALT;
  RETURN 0;
END;
RETURN a/b;
ENDROUTINE div;

The following example illustrates using directives with BEGIN–END blocks.

Example

BEGIN (TIMESCALE = PS)
p = (1) a + b;       // time units are PS, arithmetic is {US}
BEGIN (TC, TIMESCALE = MS)
q = (1) c + d;       // time units are MS, arithmetic is {TC}
END;
r = (2000) e + f;   // time units are PS, arithmetic is {US}
END;

2.6.2 IF–THEN, IF–THEN–ELSE

Execute a statement conditionally.

The IF–THEN statement executes a statement if a test_condition is true.
If the test_condition is false, control passes to the statement following the IF–THEN statement.

The IF–THEN–ELSE statement executes one statement if a test_condition is true, and an alternative statement if the test_condition is false. There is no semicolon terminating the statement preceding the ELSE keyword.

The test_condition is a behavior language expression that can include
routine calls. The test_condition must produce a twostate result that is
ture when the low-order bit is a one (1); false when the low-order bit is a
zero (0).

The test_condition must evaluate to a 0 or a 1. If a fourstate port or
state is used in the test_condition, an implicit fourstate to twostate
conversion and assignment takes place before the condition is evaluated.
The CONVxx modifiers declared on the state or port govern the conversion
process. If no CONVxx modifiers are declared, U's and Z's are changed to 0 by default.
Behavior Primitives

Syntax

IF test_condition THEN statement
IF test_condition THEN statement ELSE alternative_statement

Example

STATE ror<31:0>;
IF ror THEN
  %PRINT ('true when low bit is 1');
IF ror EQL 1 THEN
  %PRINT ('true only when ror is 1');
IF b GTR a THEN
  %PRINT ('Largest value b<\textgreater');
ELSE
  %PRINT ('Largest value a<\textless');

2.6.3 FOR

Executes a statement repeatedly based on the value of an automatically incremented or decremented control variable. Executes a statement as many times as you specify.

Syntax

FOR var_name [bit_subs] FROM start TO finish [BY step] DO statement
FOR var_name [bit_subs] FROM start DOWNTO finish [BY step] DO statement

where

var_name is a variable name that you choose. It must be declared explicitly as a state, must be twostate, and must be 32 bits or less.
start is an expression for the starting value of the var_name.
finish is an expression for the final value of the var_name.
step is an expression for a positive integer.
start, finish, and step must be twostate, and must be 32 bits or less.

Note: If it is necessary to use fourstate variables in a FOR loop, assign them to twostate temporaries before using them in the FOR statement.

The FOR statement creates loops that execute the same statement for a series of values. You must specify the starting value, finish value, and whether the value increases or decreases. You can also specify how much the value changes at each iteration. The finish and step expressions are evaluated once, before the loop starts.
Behavior Primitives

DOWNTO indicates that the value decreases with each cycle. TO indicates that the value increases. The phrase "BY step" indicates the size of each change in value. The default is 1.

Note: DECISIM uses two's complement arithmetic to increment or decrement the variable, and to compare it to the finish value.

Example

MODEL fort = ;
BEHAVIOR;
FORWARD ROUTINE loop ();
STATE x<3:0>;
ROUTINE loop ();
FOR x FROM 1 TO 15 BY 2 DO 
%PRINT ('x is ',x);
ENDROUTINE loop;
ENDBEHAVIOR;
ENDMODEL fort;

When invoked, the loop routine produces the following results:

x is 1
x is 3
x is 5
x is 7
x is 9
x is B
x is D
x is F

2.6.4 WHILE–DO, DO–WHILE

Continues to execute a statement as long as the test_condition is true. The test_condition is a behavior language expression that can include routine calls. It must produce a two-state result that is true when the low–order bit is a one (1); false when the low–order bit is a zero (0).

The test_condition cannot be an undefined or high–impedance value. If a fourstate port or state is used as the test_condition, any U#2 and Z#2 bits will be converted according to the CONVxx modifiers declared on the port or state. If no modifiers are declared, U's and Z's are changed to 0 by default. If the test_condition is an expression containing a fourstate value the conversion patterns can be specified with the CONVxx operators. If no conversion pattern is specified, U#2 and Z#2 bits are converted to 0 by default. See Appendix D for more information on CONVxx operators.
Behavior Primitives

Syntax

WHILE test_condition DO statement

DO statement WHILE test_condition

Placing the DO phrase before the WHILE phrase causes DECSIM to execute the statement at least once before it checks the test_condition.

Example

run = 1;
WHILE run DO
    BEGIN
        fetch_instruction();
        execute_instruction();
    END;

2.6.5 UNTIL-DO, DO-UNTIL

Continues to execute a statement as long as the test condition is false. The test condition must be twostate. If a fourstate port or state is used as the test condition, an implicit fourstate to twostate conversion and assignment takes place before the condition is evaluated. CONVxx modifiers declared on the state or port govern the conversion process. If no CONVxx modifiers are used, U's and Z's are changed to 0 by default. If the test_condition is an expression containing a fourstate value, the conversion process can be specified with the CONVxx operators. If no conversion pattern is specified, U#2 and Z#2 bits are changed to 0 by default.

Syntax

UNTIL test_condition DO statement

DO statement UNTIL test_condition

Placing the DO phrase before the UNTIL phrase causes DECSIM to execute the statement at least once before it checks the test_condition. The following example illustrates the difference between DO-UNTIL and UNTIL-DO.

Example

STATE x<>;
ROUTINE d ();
x=6;
UNTIL x GTR 5 DO
    BEGIN
        %PRINT (x);
        x = x + 1;
    END;
    %PRINT ('Done ',x);
ENDROUTINE d;
ROUTINE w ();
x=6;
DO BEGIN
  %PRINT (x);
  x = x + 1;
END
UNTIL x GTR 5;
%PRINT ('Done ',x);
ENDROUTINE w;

When invoked, the above routines produce the following results:

SIM> CALL d
Done 00000006

SIM> CALL w
00000006
Done 00000007

2.6.6 HALT

 Stops simulation and returns to command mode.

If the routine is executed with the ACTIVATE or SIMULATE command, HALT allows the remaining statements in the routine, and all events in the current time slot, to finish, and then returns you to command mode. You can resume execution of events following the time the HALT occurred by using the SIMULATE command.

If the routine is executed with the CALL command, HALT aborts the routine and returns you to command mode. You cannot resume executing the routine.

Syntax

HALT

Example

IF error THEN
BEGIN
  %PRINT ('Error detected');
  HALT;
END;

2.6.7 REPEAT

Repeats a statement until LEAVE or RETURN is executed.
Behavior Primitives

Syntax

REPEAT statement

Example

REPEAT
  BEGIN
  I = NEXT (I);
  IF I EQL 0 THEN RETURN;
  WAIT 100 NS;
  END;

2.6.8 SELECT

Executes one of a list of statements whose case matches the value of the select expression. SELECT is equivalent to the VAX–11 CASE instruction, and is very efficient.

Cases and select expressions are limited to 32 bits. They must be two-state. If a fourstate port or state is used as the test condition, an implicit fourstate to twoestate conversion and assignment takes place before the condition is evaluated. CONVxx modifiers declared on the state or port govern the conversion process. If no CONVxx modifiers are used, U’s and Z’s are changed to 0 by default. If the test_condition is an expression containing a fourstate value, the conversion process can be specified with the CONVxx operators. If no conversion pattern is specified, U#2 and Z#2 bits are changed to 0 by default.

Syntax

SELECT select_expression FROM
  [case, ...] : statement;
  ;
  ;
  ;
ENDSELECT;

where case is:  constant
  constant1:constant2 !a range of constants
  OTHERWISE !the keyword

where constant, constant1, and constant2 can be a number or a name declared as a constant

The select expression is evaluated once, and then compared with each case. If the case matches the select expression, its associated statement is executed.

A single constant in a case matches if the expression select_expression EQL constant is true. A range of constants in a case matches if
the expression (select_expression GEQ constant1) AND (select_expression LEQ constant2) is true.

All comparisons use two’s complement arithmetic.
Cases must be constants and must be unique. You can use multiple cases
for a single statement by separating them with commas. If any of them
match the select statement, the statement is executed. However, the
statement is only executed once, no matter how many of its cases match
the select expression.

The OTHERWISE case is optional. If no cases match the select expression
and there is no OTHERWISE case, then no statements are executed, and
execution continues with the statement following the SELECT statement.

Example

```
SELECT instr_op FROM
  [1]: ac = ac + m[mar];  !add
  [2]: ac = m[mar];       !load
  [3]: m[mar] = ac;       !store
  ...
  [240:255,0]: extended_op();  !extended operation
  [OTHERWISE]: @PRINT ("Invalid operation", instr_op, " at pc ",
ENDSELECT;
```

2.6.9 SELECTONE

Executes the first of a list of statements whose case matches the value
of the select expression. Cases do not have to be unique, and they do not
have to be constants (that is, they can be expressions). SELECTONE is
equivalent to multiple IF–THEN–ELSE–IF statements. Thus, it is slower
than SELECT, but it allows expressions in its cases.

Case expressions and select expressions can be wider than 32 bits. They
must be two-state. If a four-state port or state is used as the test condition,
an implicit four-state to two-state conversion and assignment takes place
before the condition is evaluated. CONVxx modifiers declared on the
state or port govern the conversion process. If no CONVxx modifiers are
used, U's and Z's are changed to 0 by default. If the test_condition is an
expression containing a four-state value, the conversion process can be
specified with the CONVxx operators. If no conversion pattern is specified,
U#2 and Z#2 bits are changed to 0 by default.

The following statements are equivalent:

```
BEGIN
SELECTONE x FROM
  [a] :  statement_a;
  [b] :  statement_b;
  [c1:c2] :  statement_c;
  ...
ENDSELECTONE;
```
Behavior Primitives

Syntax

```
SELECTONE select_expression FROM
   [case, ...]; statement;
   ...
ENDSELECTONE;

where case is: expression
   expression1:expression2 range of values
   OTHERWISE the keyword
```

Example

```
STATE x<63:0>;
SELECTONE x FROM
   [0:FFFFFFF#16]: %PRINT ('Bits in 32 bits');
   [100000000#16:FFFFFFFFF#16]: %PRINT ('wide data');
ENDSELECTONE;

SELECTONE executes a case statement only when

case EQL select_expression (not when the low bit of the case is true). If
you are interested only in the low bit, the following application tests for it.
```

Example

```
SELECTONE 1 FROM
   [a<0>]; ...;
   [b<0>]; ...;
   [c<0>]; ...;
ENDSELECTONE;
```

2.6.10 SELECTALL

Executes one or more of a list of statements whose case matches the value
of the select expression. Cases do not have to be unique and do not have to
be constants (that is, they can be expressions). SELECTALL is equivalent
to multiple IF-THEN statements. Thus, it is slower than SELECT, but it
can execute many statements.

Case expressions and select expressions can be wider than 32 bits. They
must be twostate. If a fourstate port or state is used as the test condition,
an implicit fourstate to twostate conversion and assignment takes place
before the condition is evaluated. CONVxx modifiers declared on the
state or port govern the conversion process. If no CONVxx modifiers are
used, U's and Z's are changed to 0 by default. If the test_condition is an
expression containing a fourstate value, the conversion process can be
specified with the CONVxx operators. If no conversion pattern is specified,
U#2 and Z#2 bits are changed to 0 by default.
The following statements are equivalent:

SELECTALL x FROM
  [a]: statement_a;
  [b]: statement_b;
  [c]: statement_c;
ENDSELECTALL;

BEGIN
  temp = x;
  IF temp EQL a THEN statement_a;
  IF temp EQL b THEN statement_b;
  IF temp EQL c THEN statement_c;
END;

Syntax

SELECTALL select_expression FROM
  [case,...]: statement;
  .
  .
ENDSELECTALL;

where case is: expression
expression:expression !range of expressions
OTHERWISE !the keyword

Example

SELECTALL 1 FROM
  [psl<0>]: %PRINT ('C set');
  [psl<1>]: %PRINT ('V set');
  .
  [psl<31>]: %PRINT ('CM set');
ENDSELECTALL;

2.6.11 RETURN

Terminates execution of a routine and returns control to the caller. An expression must be given in the RETURN statement if the routine returns a value. See Section 2.4.9.2 for information on routines that return a value.

Syntax

RETURN [ expression ]

The following model has two routines: caller and max. Caller calls max and passes two actual parameters to it. Max returns a value of six to caller.
Example

MODEL test = ;
BEHAVIOR;
  FORWARD ROUTINE caller ();
  FORWARD ROUTINE max <31:0> (arg_1<31:0>, arg_2<31:0>);
  STATE x<31:0>,
    combine<31:0>;
  ROUTINE caller ()
    x = 3;
    combine = x + max (5,6);    !value of combine is 9
  ENDROUTINE caller;
  ROUTINE max <31:0> (arg_1<31:0>, arg_2<31:0>);
    IF arg_1 GTR arg_2 THEN RETURN arg_1;
    RETURN arg_2;
  ENDROUTINE max;
ENDBEHAVIOR;
ENDMODEL test;

A routine that returns a value cannot be invoked with the ACTIVATE
statement or command.

For a routine called from the interactive command language, the
%ROUTINEVALUE system variable provides access to the return value.

Example

SIM> EVALUATE %ROUTINEVALUE

2.6.12 LEAVE

Exits from the current statement and continues execution at the statement
following the labeled statement. Statements are specified with a label,
which is declared by placing it in front of a statement in a routine.
The label must be unique within the routine. The LEAVE statement
must appear within the statement to which the label is attached. See
Section 2.1.9 for more information on labels.

Syntax

    LEAVE label_name
Example

```
ROUTINE test3();
STATE i<>;
loop: REPEAT
  stmt: BEGIN
    %PRINT (i = , i);
    IF i GEQ 4 THEN LEAVE loop; !terminates execution of "loop"
    IF i GTR 3 THEN LEAVE stmt; !terminates execution of the
    i = i + 1;
    compound statement "stmt"
  END;
END;
ENDROUTINE test3;
```

2.6.13 ACTIVATE

Schedules a routine to be executed after a specified delay. If the delay is not specified, it defaults to 0 (zero).

Syntax

```
ACTIVATE routine_name ( [ actual_parameter [, ... ] ] ) ( [ delay ] )
```

where delay is a floating point number that denotes time, a timing delay parameter passed in from the model header, or an expression. If the delay is an expression, it may contain constants, states, or ports. The expression must be in parentheses. See Section 2.9 for more information about timing delay parameters.

*actual_parameter* is an expression

Example

```
ACTIVATE ibox();
ACTIVATE task(n) {100ns};
ACTIVATE ibox() {param1};
ACTIVATE ibox() {i+1};
```

ACTIVATE evaluates the actual parameters and schedules a routine for execution. The execution then continues with the statement following the ACTIVATE statement. ACTIVATE has a default delay of zero.

ACTIVATE is not equivalent to a routine call; with ACTIVATE, routine execution continues immediately with the statement following the ACTIVATE statement, without waiting for the activated routine to execute. With a routine call, execution passes to the called routine and, when the called routine is complete, continues with the statement following the routine call.

Timing in a routine is ignored when it is called; timing is not ignored when a routine is activated.

A task is created to execute routines evoked with the behavior ACTIVATE statement, a transition on an input port declared with an ENTRY modifier, or the command language ACTIVATE command. Routines that are executed this way cannot return a value. The total number of tasks
is limited to 64. The command language SHOW ACTIVATE command displays tasks.

Routines that are called with a behavior call statement or expression, or the command language CALL command, are executed in the same task with the caller, and can return a value. The command language SHOW CALL command shows those called routines that have suspended execution.

2.6.14 %PRINT

Sends textual output to the terminal (SYS$OUTPUT), batch log file, or DECSIM log files. The default radix is 16 and is not affected by the command language SET RADIX command.

Syntax

```
%PRINT [ { print_directive [, ..] } ]
(print_element [ '{ print_directive [, ..] }' [, ..] )
```

Default

```
%PRINT (RADIX = 16, zerofill)
```

Example

```
%PRINT (radix=2, nozerosfill) ('Error detected',
   %crlf,'pc=','pc', psl='', psl);

%PRINT (%char (terminal_output_buffer));
```

The %PRINT statement outputs text to the terminal (SYS$OUTPUT) or a batch log file by default, or to a DECSIM log file specified by the [LOGNAME=logname] directive.

Print directives specify how the text is printed. They are enclosed in braces ({ }). Directives placed directly after %PRINT affect all print elements in that statement. Directives placed after a print element affect only that print element. Directives affect only the %PRINT statement containing them; they do not set defaults for all %PRINT statements.

%PRINT statements cannot be nested. %PRINT output greater than 300 characters per line is truncated.

Print elements specify the data to be printed, and are discussed in Section 2.6.14.7. Print directives are WIDTH, RADIX, LOGNAME, NOCRLF, ZEROFILL, and NOZEROFILL, and are discussed in the following sections.
2.6.14.1 WIDTH=integer
Specifications the character width of the value of the variable or expression. The integer can be 1 to 256, and can be specified with an expression or a constant. By default, width is as wide as is necessary to print the value of the variable or expression in radix 16. Width is increased as necessary to display values in other radices.

2.6.14.2 ZEROFILL
Aligns the value of an expression on the right and supplies leading zeros. ZEROFILL is the default. If the %PRINT element is not an expression, DECSIM ignores ZEROFILL.

2.6.14.3 NOZEROFILL
Shuts off ZEROFILL. Does not align the value of an expression on the right or supply leading zeros.

2.6.14.4 RADIX=integer
Specifies the radix for the print element. If the print element is not an expression, DECSIM ignores the RADIX directive. The RADIX can be any integer from 2 to 16, and overrides the default radix for the %PRINT statement. The integer can be specified with an expression or a constant. The default radix is 16.

2.6.14.5 LOGNAME=(logname_list)
Specifies one or more log files to which the print element is directed. If the logname has been previously set up with a Command Language SET LOG command, output is routed to this log file. Otherwise, an implicit SET LOG command is executed, which opens a log file. The log file remains open until you explicitly close it with CANCEL LOG. Multiple lognames can be used to print to multiple files by enclosing the list in parentheses. The lognames are separated by commas. If you specify just one logname, you can omit the parentheses.

Example

```plaintext
%PRINT (LOGNAME = (record,test, patter)) ('the result is ',a);
%PRINT (LOGNAME = porter) ('b = ',b);
```

2.6.14.6 NOCRLF
Inhibits the default carriage return/line feed (CRLF) that is normally output after the last print element of each %PRINT statement.

If you use the NOCRLF directive, you can still insert CRLF's in the output by using the %CRLF print element or the string ' ^MAJ ', which is interpreted as CTRL/M [CR],CTRL/J [FF].

2.6.14.7 Print Elements
Print elements are the data to be printed. They include strings, expressions, and functions. These are described in the following section.
%CHAR(expression)
Interprets the low eight bits of the expression as ASCII code and prints the
equivalent ASCII character. For example, the following print statement
rings the terminal bell: %PRINT (%CHAR(7));

%NET_NAME(identifier)
Prints the full hierarchical name of the given identifier. Currently, the
identifier must be a port, state, or synonym. This feature is useful for
printing the instance label and name of a port, state, or synonym when
you have several instantiations of the same model.

%DATE
Prints today's calendar date using the format dd-MMM-yyyy.
Example: 12-DEC-84

%DAYTIME
Prints the current time of day using the format hh:mm:ss.
Example: 17:05:00

%CRLF
Inserts a carriage return/line feed into the print output, so that the next
print element is printed on a new line.

%ERASE_SCREEN
Erases the screen and places the SIM> prompt and the cursor at the top
left of the screen.

%HOME
Places the SIM> prompt and the cursor at the top left of the screen.

%POSITION (X,Y)
Places the string at the location specified by the X and Y coordinates. For
example, %PRINT (%POSITION (6,2), 'This is a test'); positions the
text string so that it begins 6 lines from the top and 2 columns from the
left. Expressions can also be used.
Example: %PRINT (%POSITION (X+4, Y/2), 'What's up, Doc?');

%READ_OUTPUT_PORT
Prints the value of an output port.
Example
%PRINT ('c2 = ', %read_output_port(c2));

%TIME
Prints the current simulation time.
Example: 170 ns

Strings
Prints text literally. Strings can be as long as 256 characters. See
Section 2.1.4 for more information on how to quote strings.
Expressions
Prints the value of the expression as a number in the format specified by
the WIDTH, RADIX, and ZEROFILL directives. Known bug: the operands
in the expression cannot all be literal numbers.

2.6.15 WAIT

Suspends execution for a specified amount of simulation time, or until a
condition becomes true or false. The WAIT statement affects only the task
containing it. See Section 2.6.13 for more information on tasks.

WAIT returns a value of 0 if the test condition becomes true at or before
the specified time, and a value of 1 if it waits the specified time. You can
specify the value of time as an integer, as an expression, or as a delay
parameter if the parameter is defined in the model header. You can also
use the keyword FOREVER. If the delay is an expression, it can contain
constants, states, or ports. The expression must be in parentheses.

Syntax

WAIT time
WAIT time UNTIL test_condition
WAIT time WHILE test_condition

where time can be an integer, a delay parameter, an expression, or the
keyword FOREVER. See Section 2.9 for more information on timing delay
parameters.

where test_condition is a twostate expression whose low bit is 1 when true,
0 when false. The test_condition cannot contain routine calls.

Local states do not trigger the evaluation of a WAIT conditional expression
(WAIT WHILE and WAIT UNTIL) because local states are dynamically
allocated and cannot be changed by other routines or by the command
language. The state will never change, and the test_condition will never
be satisfied. Triggering of these expressions is limited to the other states
in the expression (global states or local states declared as STATIC).

Example

WAIT FOREVER WHILE (in EQL 0);
WAIT 3.7 WHILE (ex1 EQL ex2); "default time scale is ns
timeoutflag = WAIT 4us UNTIL ackflag; !captures WAIT return value
WAIT 10 ns;
WAIT (2+1);
WAIT pal; "pal is a timing delay parameter
WAIT FOREVER UNTIL (static_local) !STATIC local state is allowed
Behavior Primitives

WAIT FOREVER UNTIL (local_state); !expression will
!never be triggered

See Section 2.8.4.5 for more information on time units.

2.6.16 Assignment

An assignment is an expression that takes the value from the right side of the equal sign (=) and places it in the variable on the left side of the equal sign. You can assign values to variables in two ways: with delayed assignments or direct assignments.

Delayed assignments, which can have delays of zero (0) or more, are scheduled, and can trigger routines. Ports require delayed assignments. Global states can be assigned with either delayed or direct assignments, except for arrays, which require direct assignments. See Section 2.4.2 for more information on making assignments to arrays.

Both delayed and direct assignments can trigger watches; however, only delayed assignments can trigger entry routines.

You can delay the assignment of a value to a port or global state in three ways: by specifying a time, a delay parameter, or an expression. To use a delay time, enclose the time, which is an integer or a floating point number, in curly braces ({})) and place it after the equal sign (=). To use a delay parameter, enclose the parameter name in curly braces. To use an expression, enclose the expression in parentheses and enclose the parenthetical expression in curly braces.

By specifying a delayed assignment, you are scheduling the assignment. See Section 2.9 for more details on timing delay parameters.

Syntax

name[[word_sub][<bit_sub>]] = expression
name = (delay) expression

where word_sub is a word subscript,
    bit_sub is a bit subscript,
    delay is a delay parameter, a time, or an expression

Example

OF = {30} 10
bus = {10.7 us} 0
bus = {pal} 2         !pal is defined in the model header.
bus = {(xti + yti)} 3  !xti and yti can be states or ports.

The delay time affects only the assignment of the value—it does not affect any operation on the right side of the equal sign. In the following example, the final value of y is 3.
Example

\begin{align*}
x &= 3; \\
y &= (10)x; \\
x &= (5)2; \\
\end{align*}

Direct assignments bypass the scheduler and do not activate entry routines. Local states require direct assignments.

The following example illustrates direct and delayed assignments.

Example

\begin{align*}
A &= 0; & \text{!direct assignment to } A \\
A &= (5)1; & \text{!delayed assignment to } A \\
B &= A; & \text{! } B = 0 \\
\text{WAIT } 5; & \text{!WAIT delays execution of next statement} \\
B &= A; & \text{! } B = 1 \\
\end{align*}

See Section 2.4.1.1 for more information on assigning values to ports and on making multiple assignments to behavior outputs. See Section 2.4.2.2 for more information on assigning values to states.

Note: You can schedule a delayed assignment only to a complete variable. For example, given STATE temp<3:0>, you cannot schedule "temp<2:1> = (10) 11". In a delayed assignment, you must assign values to all bits simultaneously.

2.6.17 Routine Call

Behavior routines and routines written in other languages such as C, PL1, and Pascal can be called from a Behavior routine. Routines are called by specifying the routine name and any actual parameters. Routines must be declared with EXTERNAL ROUTINE, FORWARD ROUTINE, or the routine itself before they are referenced.

The compiler recognizes a routine call by the parentheses (). You must include them in a routine call, even if there are no parameters. See Section 2.6.13 for an explanation of the difference between calling a routine and activating a routine.

Calls of routines that return a value can be used as expressions.

Syntax

\begin{verbatim}
routine_name( [ expression, ... ] )
\end{verbatim}
Example

The first routine call example is a statement; the second example illustrates using a routine call as an expression.

```
PRINT_STATUS();
PLA_OUT = EVAL_PLA(PLA_IN); !EVAL_PLA is a routine;
!PLA_IN is an actual parameter.
```

### 2.6.17.1 Calling a Foreign Language Routine

Your model can call a routine written in another language. Other languages, such as Pascal and C, can be used for making system service calls, calculating floating point arithmetic, manipulating strings and linked lists, or calling other programs.

To call a routine written in another language, follow this procedure:

1. Use the global declaration INCLUDE in the model to declare the .OLB or .OBJ file that contains the nonbehavioral code. For example: INCLUDE file.OLB;

2. Use an EXTERNAL ROUTINE declaration with the INCLUDED modifier to declare each nonbehavioral routine to be called. The nonbehavioral routine cannot have the same name as any routine in your model. The INCLUDED modifier indicates to DECSIM that the routine is not in this model and therefore cannot be compiled. See Section 2.4.12 for more information on the EXTERNAL ROUTINE declaration.

3. Call the routine by entering the name of the routine followed by parentheses. The %LOC() function allows you to access the actual address of a behavior data structure (see Section 2.6.17.2 for more information).

The following is an example of a behavior model that calls a Pascal routine.

**Example**

```
MODEL pascal_routines =
BEHAVIOR;
INCLUDE [lowell]pascal_routine_1.obj;
EXTERNAL ROUTINE random INCLUDED (dummy_1<31:0>); !Non-BDS routines
       !must be declared.
ROUTINE main;
random(%LOC(r_number));
%PRINT('random number = ',r_number);
ENDROUTINE main;
ENDBEHAVIOR;
ENDMODEL pascal_routines;
```

**Note:** If you instantiate more than one behavior model in a structural model and these behavior models call routines in a nonbehavior language module, you should know that DECSIM does not
keep a separate copy of the values of global variables in the nonbehavioral module for each behavior model.

A call from one behavior model to a routine in a nonbehavioral language module that references a global variable can change the value of that variable for all behavior models that call routines in the nonbehavioral module.

2.6.17.2 Accessing DECSIM Data Structures from a Foreign Routine
To access DECSIM data structures from the foreign routine, you can use one of two functions: %LOC() and %PTR().

%LOC returns the address of the actual data if it is two-state and 32 bits or less. If the data is four-state or wider than 32 bits, the address contains a pointer to the data.

%PTR returns the address of the state or port descriptor. This descriptor is several words long and contains information that includes the state or port type, width, and convert bits.

Note: %PTR and %LOC return a long word (32 bits) address. This is why your nonbehavior code should be declared with arguments that are 32 bits wide.

Contact ELSIE::DECSIM_SUPPORT for more information on %PTR and %LOC. See Section 2.6.17.3 for more information on writing foreign language routines.

Example
The example below calls a Pascal routine that generates a "random number" by getting the system time through a system service call. The program uses the system time as a seed. Both the Behavior model and the Pascal routine are shown.

MODEL random = ;
!Behavior model
BEHAVIOR;
EXTERNAL ROUTINE rnd INCLUDED (dummy_1<31:0>);
INCLUDE rnd.obj;
STATE r_number<31:0>;
ROUTINE main;
  rnd(%LOC(r_number)); !Passes the address of the
  %PRINT('random number = ', r_number);
ENDROUTINE main;
ENDBEHAVIOR;
ENDMODEL random;
This Pascal program illustrates how to write a function or procedure for use with a Behavior model.

```
MODULE pmain(output);
GLOBAL PROCEDURE rnd(var x:integer);
    var time_pc: packed array [1..11] of char;
    time_c: array [1..11] of char;
    time_i: array [1..11] of integer;
    seed: integer;
    i: integer;
begin
    TIME(time_pc);
    UNPACK(time_pc, time_c, 1); (*Where 1 is the starting subscript of time_i*)
    for i:= 1 to 11 do time_i[i]:= ord(time_c[i])-48;
    (* 12345678901 *)
    (* hh:mm:ss.th *)
    (* so leave out [3], [6] and [9] *)
    seed:= 0;
    seed:= time_i[1]+time_i[10]*10+time_i[8]*100+time_i[7]*1000;
    seed:= seed = time_i[5]*6000+time_i[4]*6000+time_i[2]*360000+time_i[1]*360000;
    seed:= seed * 192351013; (* a very large prime number *)
    if (seed < 0) then seed:= seed * (-1);
    writeln(seed);
end;
```

### 2.6.17.3 Writing Foreign Language Routines

When you write a foreign language routine, follow these guidelines:

- Indicate that the foreign routine is a module, not a program. For instance, in Pascal substitute the keyword MODULE for PROGRAM. See the example of a Pascal routine in Section 2.6.17.2.

- Indicate that the routine can be accessed by another program. In Pascal, the notation [GLOBAL] in front of the procedure or function header indicates this accessibility. For example: [GLOBAL] PROCEDURE rnd (VAR x: INTEGER);

### 2.7 Expressions

Expressions do arithmetic and logical calculations. They may consist of operators and operands, or just a single operand. There are two types of expressions:

- **Numeric.** These expressions can be twostate, fourstate, or mixed twostate and fourstate. They can be from 1 to 256 bits wide.

- **Time.** These expressions contain no operators, just literals.

The width of the result of the expression is determined by the widest operand, unless you explicitly specify width with the WIDTH directive. See Section D.3 for more information.

The order of evaluation for routine calls and assignments in expressions is undefined. For example, in the statement `a=r()+s()`, it is uncertain whether routine `r` or `s` will be called first.
The following sections describe the various types of operators and operands.

### 2.7.1 Operands

Operands contain the values in an expression. Operands used in expressions include:

- Variables – states, ports, formal arguments, and synonyms. Some of these can have bit and word subscripts.
- Constants – names that are bound to numbers at compile time.
- Literals – numbers.
- Routine calls – calls of routines that return a value.

### 2.7.2 Operators

The behavior language operators and their precedence are described in Appendix D.

The following sections describe the syntax and give examples of the various types of expressions.

### 2.7.3 Names

Operands (the simplest form of an expression) are names that specify states, ports, formal arguments, constants, and synonyms.

**Syntax**

```
name [word_subscripts] [bit_subscripts]
```

**Example**

```
IF temp<2> THEN %PRINT ('value is 1');
```

The optional subscripts don’t apply to all classes of names. See Section 2.1.6 for more information on using subscripts.

### 2.7.4 Literals

Expressions can be literal numbers.

**Syntax**

```
digits [%radix]
```

See Section 2.1.8 for more information on radix.
Behavior Primitives

Example
AF#16
1011#2
13

2.7.5 Unary Operations
Expressions can consist of operations on one expression.

Syntax
unary_operator [directives] expression

Example
NOT x
NOT (-3)

2.7.6 Binary Operations
Expressions can consist of operations on two expressions.

Syntax
equation binary_operator [directives] expression

Example
a + 3 * c
(a + 3) * c
a + (%carryin = ci) b

2.7.7 Routine Call
Expressions can be routine calls.

Syntax
routine_name ([expression [, ...]])

Example
y = f(x);
IF compare() THEN HALT;
The second example calls a routine named compare, which must return a value. See Section 2.4.9.2 for more information on routines that return a value.
2.7.8  \textbf{WAIT}

Discontinues execution of a routine and schedules it to resume after a specified amount of time or after a specified condition becomes true. See Section 2.6.15 for more information.

2.7.9  \textbf{Nested Expressions}

One expression can be nested in another. Order of evaluation is determined by operator precedence, unless parentheses are used. By enclosing an expression in parentheses, you can:

- Override ordinary precedence. For example, \((a+b)*c\) causes DECSIM to perform the addition operation before multiplication.

- Extract bits from an expression. For example, \((a + b)<7:3>\).

Parenthesized expressions can also extract bits from constants, which don't have subscripts. The right-hand (least significant) bit is assumed to be bit number 0; the left-hand (most significant) bit number is assumed to be one less than the number of bits in the expression. See Section 2.1.6 for more information on subscripts.

Example

\begin{verbatim}
CONSTANT n = 17304#16;
\texttt{x} = \texttt{(n)<15:13>};
\end{verbatim}

2.8  \textbf{Directives}

Directives give DECSIM additional information on how to perform certain operations. They are enclosed in braces \{ \\} . Multiple directives are included within one set of braces and are separated by commas. The following sections describe directives and their application.

2.8.1  \textbf{Scope of Directives}

Directives can be specified for the entire behavior model, or for smaller pieces of that model. Directives at a lower level overrule those at a higher level.

Example

\begin{verbatim}
MODEL x = ;
BEHAVIOR (TC);
\texttt{!behavior subblock default is TC}
STATE a<>\texttt{, b<>\texttt{, c<>};}
ROUTINE \texttt{x () (US)}; \texttt{!routine default is US}
BEGIN (TC); \texttt{!block default is TC}
c = a \texttt{+ (US) b}; \texttt{!this addition is US}
END;
ENDROUTINE x;
ENDBEHAVIOR;
ENDMODEL x;
\end{verbatim}
2.8.2 TIMESCALE

Specifies the default units for time. It can be specified for the entire model, or for smaller pieces of that model.

**Syntax**

\[
\text{TIMESCALE} = \text{units}
\]

where units are FS, PS, NS, US, MS, S.

**Example**

\[
\text{BEHAVIOR } \{ \text{TIMESCALE} = \text{PS} \};
\]

See Section 2.8.4.5 for more information on time units.

2.8.3 MODULES

Specifies modules that are accessed by the model. The MODULES directive is placed in the behavior header. See Section 2.3 for more information on using the MODULES directive.

**Syntax**

\[
\text{MODULES} = \{ \text{module}\_\text{name} [, \text{module}\_\text{name}, ...] \}
\]

where module\_name is any legal VMS file specification. The default extension is .SLB.

**Examples**

\[
\text{BEHAVIOR } \{ \text{MODULES} = (\text{inbox, ebox.slb});}
\]

\[
\text{BEHAVIOR } \{ \text{MODULES} = (\text{mymodel:inbox, disk$:[user1.decsim]mbox});}
\]

2.8.4 Operator Directives

The following sections describe the directives that modify operators.

2.8.4.1 WIDTH

Specifies the width of the result of an operation. WIDTH must be at least 1 and less than or equal to 256. If a result is wider than specified with WIDTH, the most significant bits are truncated. If the result is narrower than specified, it is zero–extended if an unsigned operation or sign extended if a two's complement operation.
Behavior Primitives

Syntax

\[ \text{WIDTH} = \text{literal} \]

Example

\[
\begin{align*}
\text{w32} &= 1111 \#2 + 1\#2 ; & \text{!without WIDTH, result is four bits;} \\
\text{w32} &= 1111 \#2 + (\text{WIDTH}=5) \ 1\#2 ; & \text{!result is five bits;} \\
\text{w64} &= \text{w32} \times \text{w32}; & \text{!very slow} \\
\text{w32} &= \text{w32} \times (\text{WIDTH}=32) \text{ w32}; & \text{!fast - but truncates the answer.}
\end{align*}
\]

\[ \text{2.8.4.2} \quad \%\text{CARRYIN} \]

Allows you to specify a 1-bit carry input to an addition or subtraction. \%\text{CARRYIN}, when used with \%\text{CARRY} and \%\text{OVERFLOW}, lets you capture the overflow and carry-out bits of a three-way sum in a single operation.

Syntax

\[ \%\text{CARRYIN} = \text{expression} \]

Example

\[
\begin{align*}
\text{s} &= \text{a} + (\%\text{CARRYIN} = \text{c}) \text{ b}; & \text{!adding a + b + the carry in c} \\
\text{! The following example adds a + b + the carry in c, and also} \\
\text{! stores the carry out and overflow bits} \\
\text{s} &= \text{a} + (\%\text{CARRYIN} = \text{c}, \text{ co} = \%\text{CARRY}, \text{ ov} = \%\text{OVERFLOW}) \text{ b};
\end{align*}
\]

\[ \text{2.8.4.3} \quad \%\text{CARRY} \]

Stores the carry or borrow bits resulting from the operation. It provides access to the carry bits for all bit positions, not just the most significant bit. Thus, it is as wide as the widest operand.

Syntax

\[ \text{name[word_sub]} [\text{bit_sub}] = \%\text{CARRY} \]

Example

\[
\begin{align*}
\text{s} &= \text{a} + (\text{ co} = \%\text{CARRY}) \text{ b} & \text{!the carry is stored in "co"} \\
\text{s} &= 11\#2 + (\text{co} = \%\text{CARRY}) \ 11\#2 & \text{!the carry is 11\#2}
\end{align*}
\]

The following example shows using \%\text{CARRY} to calculate the borrow bits in a subtraction.

\[
\begin{align*}
\text{b} &= 100000\#2; \\
\text{c} &= 11111\#2; \\
\text{a} &= \text{b} - (\text{TC}, \text{ d} = \%\text{CARRY}) \text{ c};
\end{align*}
\]

In the above example, \text{d} has a value of 11111\#2.

Doing the following addition produces carries, which are the complements of the borrow bits produced by the above example.

\[
\text{a} = \text{b} + (\text{TC, d_prime = \%CARRY, \%carryin=1}) (\text{NOT} \text{ c});
\]

2–61
Behavior Primitives

See the following section for an example of what is captured by %CARRY and %OVERFLOW.

2.8.4.4 %OVERFLOW

Used with two's complement arithmetic, %OVERFLOW stores the overflow bits resulting from the operation. %OVERFLOW provides access to the overflow bits for all positions, not just the most significant bit. Thus, it is as wide as the widest operand.

An overflow occurs when you add two positive numbers and get a negative answer, or add two negative numbers and get a positive answer. In other words, an overflow occurs when you wrap around the range of values allowed within the fixed-precision number system.

Syntax

\[
\text{name[word_subs] [bit_subs]} = \%\text{OVERFLOW}
\]

Example

\[
\text{STATE } a<31:0>, b<31:0>, ov<31:0>;
\]
\[
s = a + \{ \text{ov} = \%\text{OVERFLOW} \} b ! \text{The overflow is stored in } "\text{ov}".
\]

The following example shows what is captured by %CARRY and %OVERFLOW.

\[
\text{STATE}
\]
\[
a<6:0> = 0110101\#2 ;
b<6:0> = 0011101\#2 ;
c = a + \{ \text{co} = \%\text{CARRY}, \text{ov} = \%\text{OVERFLOW} \} b ;
\]
\[
! 0110101 \text{ (a)}
! + 0011101 \text{ (b)}
! \text{ ------}
! 1010010 \text{ (c)}
! \text{ co = 0111101\#2}
! \text{ ov = 1000111\#2}
\]

2.8.4.5 Time

Specifies delays (with WAIT, ACTIVATE, and delayed assignments). The time must be in curly braces (\{\}). Expressions can also be used for time values. The expression must be in parentheses. The space between the digit and the time unit is optional. The unites of time are:

- FS (femtoseconds)
- US (microseconds)
- PS (picoseconds)
- MS (milliseconds)
- NS (nanoseconds)
- S (seconds)
NS is the default time unit. You can change the default with the TIMESCALE directive. See Section 2.8.2 for more information.

Syntax

digits [. digits] [time_units]
[. digits [time_units]]

Example

\[ x = (3.7 ) 5; \]
\[ y = (4 fs) 4; \]
\[ \text{WAIT .5 ps;} \]
\[ \text{WAIT ((x+y)*b));} \quad \text{!delay is specified by an expression which must be in parentheses} \]
\[ \text{ACTIVATE task (68);} \]

2.8.4.6 Arithmetic Representation

Specifies whether operations are performed using TC (two's complement) or US (unsigned) arithmetic. The default is US.

Syntax

TC
or
US

Be aware of sign extension when using TC. For example, given:

\[ \text{STATE X<31:0>, Y<31:0>;} \]
\[ X = Y + 1\#2; \]

1#2 is 1 bit wide and Y is 32 bits wide. The 1#2 has to be sign-extended to 32 bits before the addition. The high-order bit of 1#2 is 1, so the sign is 1. Thus, 1#2 is extended by 31 1's, yielding FFFFFFFF#16, which is -1 in two's complement. Thus the above statement is equivalent to \[ X = Y + FFFFFFFF#16 \], so \[ X \] is 1 less than \[ Y \], not 1 greater than \[ Y \]. If you want to represent 1#2 as a positive number, use 01#2.

2.9 Timing Delays

Timing delays in Behavior models are modeled as transport delays; therefore, spikes are not filtered out. The specification for a delay can be a literal, an expression, or a parameter that is passed in from the model header.

2.9.1 Literals

Literals can denote delays in WAIT statements (see Section 2.6.15), ACTIVATE statements (see Section 2.6.13), and in assignments (see Section 2.6.16).
2.9.2 Expressions

Expressions can denote delays in WAIT statements (see Section 2.6.15), ACTIVATE statements (see Section 2.6.13), and in assignments (see Section 2.6.16).

Expressions must be enclosed in parentheses. Expressions can contain global states, local states, ports, or constants.

Syntax

WAIT (time_expression);
ACTIVATE routine_name (time_expression);
name [[word_sub]] [[bit_sub]] = (time_expression) expression;

where time_expression in its simplest form can be a single variable or, in its most complex form, a polynomial.

Example

MODEL add o<0> = i<0>;
BEHAVIOR;
STATE b<15:0>, c<15:0>, d<15:0>, e<15:0>, f<15:0>;
CONSTANT g = 20;
ROUTINE add1();
    STATE h<15:0>;
    b = 5;
c = 8;
b = {(i)}1;       // port
    c = {(b)}3;       // global state
    d = {(b+1)}4;     // global state + literal
    e = {(b+1)(b*e)/i}; // polynomial
    f = {(c+g)}5;     // global state + local state
    f = {(g)}10;      // constant
ENDROUTINE add1;
ENDBEHAVIOR;
ENDMODEL add;

2.9.3 Formal Delay Parameters

In the Behavior Language, you can define formal delay parameters in the model header after the model name, or pass a value to the parameter with the Network Description Language. The number of delay parameters that you can define in the model header is unlimited.

Syntax

MODEL modelname [parameter_name [= parameter_value], ...] [outputs]
    = [inputs];

where parameter_value is an integer or a floating point number.

In the first example below, the parameters are defined in the model header; in the second example, the parameter values will be passed in from the network description file.
Examples

```plaintext
MODEL adder [p1=1, p2=2] c<0>, out<0> = cin<0>, a<0>, b<0>;
MODEL adder [p1, p2] c<0>, out<0> = cin<0>, a<0>, b<0>;
```

2.9.4 Actual Delay Parameters

You can use the parameter_name as an actual parameter wherever a timing delay is needed in a routine, in WAIT statements (see Section 2.6.15), ACTIVATE statements (see Section 2.6.13), and assignments (see Section 2.6.16).

Example

```plaintext
ROUTINE add ();
  out = (p1) a + (temp=%CARRY, %CARRYIN=cin) b;
  WAIT p2;
  c = (p1) temp;
  ACTIVATE sub() (p2);
ENDROUTINE add;
```

You can override parameter delays when you instantiate the model with the Interconnect Language. You can also pass a value to a formal parameter that has not been defined. See Section 3.8 for more information about overriding parameters and passing parameters with the Interconnect Language.

2.10 Scheduling and Sequencing Statement Execution

This section describes how to order execution in a behavior model.

2.10.1 How the Scheduler Works

Times specified in a behavior model are relative to the current time. There are no absolute times. DECSIM executes events in the following order:

1. By time (often called event-driven simulation). All delayed assignments are scheduled, and are executed at the specified time. It is the assignment, or transfer, of data that is delayed; the calculation or evaluation of the right side of the assignment is performed immediately. See Section 2.6.16 for more information on assignments. When more than one event is scheduled for the same time, the following priorities apply:
   a. Changes for ports and global states occur.
   b. Entry routines are invoked.

2. By sequence (often called non-event-driven simulation). Routines are executed in the order in which they are called. Statements in routines are executed sequentially except as modified by conditional, loop, leave, and return statements.
Behavior Primitives

Example

```
ROUTINE a ();
   x = (10) 1;
   y = (10) 0;
   ACTIVATE r (); (10);
       z = (10) 0;
ENDROUTINE a;

ROUTINE r ();
   z = (0) 1;
ENDROUTINE r;
```

!assuming routine starts at time 0,
!x scheduled to have value 1 at time 10
!y scheduled to have value 0 at time 10
!routine r scheduled to be activated at
!time 10
!z scheduled to have value 0 at time 10
!the final value of z is 1

Assuming that a runs at time 0, the sequence of events for the above example is:

1. x goes to 1 at time 10
2. y goes to 0 at time 10
3. z goes to 0 at time 10
4. r starts at time 10
5. z goes to 1 at time 10

Multiple routines scheduled to be activated in the same time slot may give unexpected results. Multiple routines scheduled for the same time slot are executed in the order they were scheduled. However, when routines are activated by signal changes, it is not always possible to determine which routine was scheduled first.

2.10.2 Timing and Sequencing Statement Execution

You can control the order of execution of statements in several ways:

- Make the order of the statements in the routine the same as the order in which the hardware performs the actions being modeled.
- Use conditional statements, which are triggered by an expression or variable used as a flag, to execute statements or call other routines.
- Use loop statements to execute statements repeatedly.
- Spawn parallel tasks with the ACTIVATE statement. See the description following this list for more information on tasks.
- Synchronize execution to a specific time or signal change with the WAIT statement.
- Specify delays in assignments to ports and states.
- Simulate from the interactive command language with the CALL, ACTIVATE, WATCH, and SIMULATE commands.
3 Interconnect Language

The interconnect language connects primitives. It allows you to build models made up of primitives, and in turn, to interconnect those models. The final result is a network that can be simulated with DECSIM.

Together, primitive definitions and the interconnect language comprise the models you use to describe your network.

3.1 Network Description File

The Network Description File (default extension .NET) is the source file for all models. It is written with ASCII characters, using the conventions of the network description language.

There are two formats of source files:

- Library—Compiles models into a compact form that can be easily read by NETPRO. This format cannot be loaded or simulated in DECSIM.
- Network—Top level of your hierarchy that can be loaded and simulated in DECSIM.

3.1.1 Library Format

Network Description Files written in library format define models (connect primitives) that are bracketed with the MODEL—ENDMODEL keywords. Within the MODEL—ENDMODEL keywords, you can write local model definitions, declarations, primitives, and instances of other models.

You compile these files with the DECSIM COMPILE/LIBRARY command to get a file with the .NLB extension. The models in the .NLB file can then be instantiated in any network description file.

You cannot load an .NLB file for simulation—you can only load an .NOB file (see the following section).

The following is an example of a source file written in library format.

```
MODEL knot out = in;
EQUATION out = not in;
ENDMODEL knot;
```
3.1.2 Network (Top Level) Format

Network description files written in network format are not bracketed with the MODEL-ENDDMODEL keywords. They contain instances of models and specify how they are interconnected. In other words, they define a network. Normally, they do not contain model definitions. A model definition in such a file applies to the defined network, and is not available outside that network.

You compile these files with the COMPILE command to get a file with the .NOB extension that can be simulated in DECSIM.

Top-level files containing MOS primitives are compiled with the DECSIM COMPILE/PROCESS command.

The following is an example of a source file written in network format.

library  not_gate;
A1: knot out = in;
output out;
input in;

3.2 LIBRARY Statement

The LIBRARY statement specifies the .NLB files to be searched. These .NLB files contain the elements to be used in assembling your network.

You can place a LIBRARY statement in a Network Description File of either format.

The location of the LIBRARY statement in the network hierarchy governs the order in which DECSIM searches libraries for a model definition, when that model definition is to be instantiated.

You can specify multiple LIBRARY statements in a model. You can also specify multiple files in a LIBRARY statement.

Format

LIBRARY file [, ... ];

3.2.1 LIBRARY Statement Defaults

Your current directory is the default area.

The default extension is .NLB.

If no LIBRARY statement is given, the default is the system library. You can also specify library files with the COMPILE and CONFIGURE commands.
Example

LIBRARY MYMACS;

This example causes NETPRO to look for a library named MYMACS.NLB in your disk area.

3.2.2 Library Search Procedure

When a model is instantiated, NETPRO searches the library files specified in the LIBRARY statement to find the proper model definition to instantiate. A file list is searched from right to left. NETPRO follows these steps during a library search:

1. NETPRO searches for the most recent local definition of the model (defined at the current level).

2. NETPRO searches the most recent library referenced by the LIBRARY statement in the same block (the current level). LIBRARY statements that appear within a MODEL-ENDMODEL block or STRUCTURE-ENDSTRUCTURE block have no effect outside that block.

3. NETPRO repeats steps 1 and 2 for each level, beginning with the innermost level and continuing outward. LIBRARY statements within a subblock take priority over models and LIBRARY statements outside the same subblock.

4. NETPRO searches the default system component library if it has been defined by the DECSIM CONFIGURE command.

A model name can be defined in any number of library files. Any library file can appear in more than one LIBRARY statement. Respecifying a library file moves that file to the beginning of the search list.

A model that is defined in a network description takes priority over the same name defined in any library file. However, each model name can be defined only once at each level. The STRUCTURE or MODEL keyword opens a new network description. Thus, a model that is defined in the same source file but outside a given structure subblock can be redefined (once) within that subblock.

NETPRO continues the search until it finds a model definition with a model name matching the one in the instance statement. If a subblock name is used in the instance statement, a match for that subblock name in the model definition must also be found. If no model with that name exists, an error message appears.
Interconnect Language

Example 1
Given a model definition as follows:

MODEL OUTER ...
MODEL ALU1 ...
... ! Definition #1
ENDMODEL ALU1;
LIBRARY 'FILEC','FILED';
MODEL INNER ...
LIBRARY 'FILEB';
MODEL ALU1 ...
... ! Definition #2
ENDMODEL ALU1;
INST1: ALU1 ...
ENDMODEL INNER;
ENDMODEL OUTER;
The search order for the model definition of ALU1 to be instantiated at INST1 is:
1 Local definition #2 of ALU1
2 File 'FILEB'
3 Local definition #1 of ALU1
4 File 'FILED'
5 File 'FILEC'
6 System library

Example 2
! File A.NLB contains models t, u, and v.
! File B.NLB contains models t, u, and w.
LIBRARY a; ! Models t, u, v are now known.
MODEL u outs = ins; ... ! This redefines 'u'
! In here, any model from library A is valid, INCLUDING model u.
ENDMODEL u;
! Here, you cannot define model u again (can't define it twice
! on the same hierarchical level)
LIBRARY b; ! Model t from B.NLB replaces t from A.NLB.
! Model u from B.NLB is ignored because the
! network contains a 'local' definition for u.
MODEL x outs = ins;
STRUCTURE y;
! In here, you can redefine model u.
LIBRARY a; ! Models t and u from A.NLB replace t from
! B.NLB and u from the 'local' definition
! because STRUCTURE y opens a new network
! description.
MODEL z outs = ins; ! Another sample of a local model definition
... ENDMODEL z;
...
ENDSTRUCTURE y;
! Here, the definition of model z is thrown out. The definition of
! model t goes back to file B.NLB and the definition of model u
! goes back to the 'local' definition above.
... ! More subblocks allowed here
ENDMODEL x;
! Known models here are t (B.NLB), u (local), v (A.NLB), w (B.NLB),
! and x (local).
3.3 Model Definition

The format for defining a model is as follows:

```
MODEL modelName [parameters] outputs = inputs ; !Model header
[Signal declaration section]
Model body (structure subblocks)
ENDMODEL modelName;
```

The following section further explains the elements of the model definition:

- **MODEL**: Keyword that identifies the statement and the entire element specification as a model definition.
- **modelName**: Name that will be used in the instance statement to call this model.
- **[parameters]**: List that defines the formal parameters that can be used inside the model definition.
- **outputs = inputs**: List of output signals separated from the list of input signals by an equal sign.
- **global declarations**: Optional signal declarations that are enclosed in GLOBAL ENDGLOBAL brackets, assign values, or turn off warning messages, and have scope over all subblocks in the model body.
- **model body**: A set of structure subblocks.
- **ENDMODEL**: Keyword that ends the current level of model definition. The model name following the keyword ENDMODEL must match the model name on the MODEL keyword.

3.3.1 Model Definition Header

The model definition header defines the name, optional formal parameters, and any input or output connections. The model name can be any combination of 0–9, a–z, the percentage sign (%), and the underscore (_). Lowercase letters are converted to uppercase. If special characters are needed, the name can be enclosed in single quotation marks. A model name can contain up to 127 characters.

3.3.1.1 Formal Parameters

Parameters are defined (and usually assigned default values) in the formal parameter list in the model header. Parameters are enclosed in square brackets. The formal parameter list must be name–based.
Interconnect Language

Format

{ parametername = default_value [ , ... ] };

Parameter names can contain letters, numbers, underscore (_), and the percentage sign (%). Any other characters must be enclosed within single quotation marks. If a parameter name begins with a number, it must be enclosed in quotation marks. Parameter names can have up to 132 characters.

If no parameters are to be passed, the list and its enclosing brackets can be omitted from the definition statement. A space (or tab) must then separate the model name from the signal list:

MODEL 74XXX  4/2 = 1/A, 2/B, 3/C;

A formal parameter can appear with or without a default value, and these two forms can be mixed in the same parameter list. See Section 3.8 for more information on parameters.

3.3.1.2 Connection Lists

The signal connection list, located in the model header, specifies the input and output signals that can be connected to other models in the network.

Format

output-signals = input-signals

There are two formats of signal connection list:

- Signal-based
- Pin-based

3.3.1.2.1 Signal-based Connection List

This is a list of signals, separated by commas. Signal names can contain letters and numbers and embedded spaces. You can use other characters by enclosing them within quotation marks. Subscripted signals are allowed. The same signal with different subscripts can be repeated in the list, although the subscript ranges cannot overlap.

Example

01, 02<3:0>, 03<2> = I1, I2<7:3>, I3<0>, I4;
! All signals are OK. Not all have subscripts.
A<0>, A<1>, A<2>, A<7:3> = I1, I2;
! OK. Signal "A" can appear multiple times if
! it has different subscripts.
A, B, C, A = I1, I2;
! ILLEGAL: signal "A" appears multiple times.
A<0>, B, C, A<0> = I1, I2;
! ILLEGAL: signal "A<0>" appears multiple times.
A<7:4>, B, A<5:0> = I1, I2;
! ILLEGAL: the range "A<5:4>" overlaps
3.3.1.2.2 Pin–based Connection List

This is a list of paired names separated by commas, where each pair has the format: pin–name / signal–name

The following restrictions apply to pin–names:

- The same characters are valid for pin–names as for signal names, although pin–names are often named with numeric characters.
- Subscripts are not valid.
- The signal–name must be only one bit wide.
- Each pin–name must be distinctive in relation to other pin–names and signal names.
- Pin–names cannot be referenced within the model definition.
- If pin–names are used in the model header, all signals must be defined in pin–name pairs.

Example

\begin{align*}
1/A, 2/B, P/C, Q/D<0> &= 3/E, R/E<1>; \quad \text{!All signals are OK. Not all signal–names have subscripts.} \\
1/A<0>, 2/A<1>, 3/A<2> &= 4/I1, 5/I2; \quad \text{!OK. Signal "A" can appear multiple times if it has different subscripts.} \\
1<0>/A, R<0>/B &= 4/I1, 5/I2; \quad \text{!ILLEGAL. Pin–names "1" and "R" cannot have subscripts.} \\
1/A<0>, 2/B<1:0> &= 3/C<7:4>; \quad \text{!ILLEGAL. Signal–names "B<1:0>" and "C<7:4>" are not one bit wide.} \\
1/A, 2/B, 1/C &= 3/D; \quad \text{!ILLEGAL. Pin–name "1" appears multiple times.} \\
A/R, B/S &= C/A; \quad \text{!ILLEGAL. "A" is defined as both a pin–name and a signal–name.} \\
1/A, 2/B, C &= 3/D, E<0>; \quad \text{!ILLEGAL: signals "C" and "E<0>" are not defined with pin–names.} \\
\end{align*}

MODEL M1 A/R, B/S = C/T, D/U;

! The following instance is valid. It references signals "R" and "T", defined in the model header.

I1: NOT R = T;

! The following instance is invalid. It references pin–names "B" and "D", instead of the defined signals "S" and "U".

I2: NOT B = D;

ENDMODEL M1;

3.3.2 Bidirectional Connections

Bidirectional signal declarations apply to two cases:

- When a model header signal connects to both inputs and outputs in the model.
- When a MOSFET source or drain is connected to a model header signal.
Bidirectional signals can appear either as inputs or as outputs (but not both) in the formal connection list. If you list bidirectional signals as outputs, DECSIM does not give any warning. If you list them as inputs, DECSIM issues a warning and assumes the signals are bidirectional. To turn off the warning message, use the BIDIRECTIONAL signal declaration. An example of a bidirectional pin is a tristate bus transceiver.

See Section 3.6.2.1 for more details on the BIDIRECTIONAL declaration.

3.3.3 GLOBAL Subblock

The optional GLOBAL-ENDGLOBAL section of a model is located after the model header, but before the first subblock inside the model. Signal declarations and statements placed here have scope over all structure subblocks within that model. For this reason, global signal declarations must only reference signals that are part of the formal connection list of the model header. If a signal declaration that is in the GLOBAL subblock references a signal that is not in the model header, an error is reported. Any signal declaration (see Section 3.6), except for SYNONYM, INPUT, and OUTPUT, can be placed in the GLOBAL-ENDGLOBAL block. If a “label.pin” is referenced by a signal declaration in the GLOBAL subblock, an error is reported.

Declarations in the GLOBAL section of a model may not be overridden in subsequent subblocks. Placing a signal declaration in the GLOBAL section is equivalent to placing a duplicate of that declaration in every subblock inside the model definition.

3.3.4 Model Body

The model body is a set of structure subblocks. The subblocks can be arranged in any order. Subblocks allow you to describe the same logical block in a different style or level of detail (for example, equation and MOS), while using the same model header. This feature permits you to compare two modeling styles closely.

When you instantiate the model, you specify model name and the subblock to be used.

3.3.4.1 Structure Subblock Components

Structure subblocks are placed in the body of your model. A structure subblock contains the following:

- STRUCTURE keyword—Starts the body of the structure subblock, which describes the network connectivity of the element (as opposed to the behavior subblock, which describes the functionality of the element). If only one structure definition is included in a model definition, the STRUCTURE and matching ENDSTRUCTURE statement can be omitted. If more than one structure definition is included in the model definition, then the STRUCTURE keyword must be followed by a subblock name identifying the particular structure definition. Only one null-named subblock is allowed.

- Subblock name—Identifies the particular subblock.

- Structure subblock body—A connective description of the model, which can also contain local model definitions.
Interconnect Language

- **ENDSTRUCTURE statement**—Ends a structure definition. The subblock name following the `ENDSTRUCTURE` keyword must match the subblock name on the matching `STRUCTURE` keyword.

**Format**

```
STRUCTURE [subblock-name];
! connected equations, model definitions,
! instances, and other statements.
ENDSTRUCTURE [subblock-name];
```

The body of the structure subblock specifies various logic elements and their interconnection. It is analogous to a schematic representation of the model where component elements and signal connections are specified. The structure subblock references signals in the model definition in which this structure subblock appears. These signals appear in the header formal connection list. The statements in the structure subblock describe elements, interconnection of elements, or attributes of signals.

These statements are:

- Model definitions
- Instance statements
- Equation statements
- MOSFET statements
- Resistor statements
- Memory definitions
- Signal declarations

Other statements, such as `REQUIRE`, `MACRO`, and `LIBRARY`, can also appear in the structure subblock.

Instances of behavior models can be contained within a structure subblock, although the behavior model definition itself must be defined in a separate file and compiled with a different compiler (SX).

### 3.3.4.2 Subblock Names

Since a model can contain any number of structure subblocks, subblock names are used to indicate which subblock is to be instantiated at an instantiation site. The subblock name follows the `STRUCTURE` keyword and the `ENDSTRUCTURE` keyword. A subblock name consists of letters, numbers, underscore (_), and percentage sign (%); or the subblock name can be null. Any other characters must be enclosed in single quotation marks. The model name length plus the subblock name length must not be greater than 126 characters.

**Note:** If there is more than one structure subblock and no name is supplied when the model is instantiated, the first subblock in the model file is used by default.

**Format**

```
STRUCTURE subblock-name;
```

Restrictions that apply to subblock names with subblocks in a model definition are as follows:

- A model definition cannot have more than one structure subblock with a null subblock name.
- Subblock names at the same definition level must be unique.
• A subblock name cannot be the keywords STRUCTURE or BEHAVIOR.

If a model definition consists of only one structure subblock and no GLOBAL section, then the STRUCTURE/ENDSTRUCTURE keywords are optional. If they are omitted, the processor generates an implicit structure subblock with a null subblock name.

3.5 Hierarchy in Model Definitions

Model definitions can be nested. In other words, if a model definition contains a structure subblock, that subblock in turn can contain a model definition. Model definitions cannot appear inside a behavior subblock.

Model instances can appear in a subblock. These models can contain instances of other models nested to any depth.

4 INSTANCE STATEMENTS

The instance statement calls (instantiates) a model definition. The instance statement consists of the name of the model to be instantiated, which matches a model name defined in the model header of the model to be instantiated, a signal connection list, and an optional label and parameter list. This instance statement can be placed in a .NET file of either library or network format.

Format

[label:] model_name [.subname] [[parameters]] signal_connection_list;

4.1 Label

The label, terminated by a colon, distinguishes the model being called from others in the network, particularly from other instances of the same model. A label consists of up to 31 letters and digits; special characters can be included if the characters or the entire label is enclosed in single quotation marks. The colon is not considered to be part of the label name and therefore should be outside the quotation marks.

In many networks, the label corresponds to the IC label, or “reference designator,” used in the network schematic, and customarily follows the same DEC standard, that is, a letter followed by a number (however, this is not a DECSIM-imposed restriction):

Example

   e4:
   e31:
   4a12:
   '98-05215':

Labels are optional; however, if you plan to use “label.pin” signal names, or if you want to interactively access hierarchical signal names in the simulator, you must provide a label.
Labels that contain spaces or special characters, such as the semicolon (;), must be enclosed in single quotes.

If you omit labels from your network description, DECSIM inserts implicit labels in the following format (where integer is any number):

$integer$

These labels should not be referenced in the model, but you can use them from the DECSIM command language to access models you left unlabeled.

Note: The $integer labels assigned by NETPRO may change each time you recompile your network.

3.4.2 Model Name

The model name is the name of the model being called (instantiated).

When a model represents an existing device model, such as a DIP or other IC, its name is usually the manufacturer's part number, for example, 7400, DC301, 12AU7, 'HI_HI'.

3.4.3 Searching for the Model Definition to Instantiate

A model definition can contain any number of subblocks. When you instantiate the model, you specify the subblock to be included. Regardless of how you specify the subblock, the model definition search is conducted according to the Library Search Procedure as described in Section 3.2.2. There are four ways to specify a subblock.

The subname specification formats are as follows:

- **modelname.subblock-name**

**Example**

```
E1: 7400.FOO [params] connections ;
```

In this case, the model definition instantiated, and the subblock selected from that model, are the first model definition found with the specified model name and subblock name.

- **modelname**

**Example**

```
E1: 7400 [params] connections ;
```

In this case, the model definition instantiated (and the subblock selected from that model) is the first model definition found that matches the given model name and has exactly one structure subblock or exactly one behavior subblock.

- **modelname.STRUCTURE**
Example

E1: 7400.STRUCTURE [params] connections ;

In this case, the model definition instantiated (and the subblock selected from that model) is the first model definition found that matches the given model name and has exactly one structure subblock.

• modelname.BEHAVIOR

Example

E1: 7400.BEHAVIOR [params] connections ;

In this case, the model definition instantiated and the subblock selected from that model are the first model definition found that matches the given model name and has exactly one behavior subblock.

3.4.4 Instance Actual Parameters

The parameter list in the instance statement allows you to pass actual parameter values into a model, overriding the default parameter values specified in that model’s header.

Actual parameters supply the values to be assigned to the formal parameters of the model being instantiated. As in the model header, the parameters are enclosed in square brackets.

The instance actual parameter list can be either name-based or order-based.

If the entire parameter list is omitted from an instance statement, the enclosing brackets must also be omitted, and a space must separate the element name from the signal list.

Example

E1: 7400 OUT = A, B ;

See Section 3.8 for more information on using parameters.

3.4.4.1 Name-based Actual Parameters

The name-based parameter list contains parameter names and corresponding values.

Format

{ parameter_name = parameter_value [ , ... ] }

The parameter name must match a formal parameter name in the header of the model being instantiated. The parameter value can be a constant, a parameter triple, or another parameter name passed in from the header of the model that contains the instance. See Section 3.8 for more information.

If parameters are omitted from a name-based list, the default value specified in the model header for that parameter is assigned.
Example

PARAM1 = 2PS, PARAM2 = 1:3:4] !PARAM1 and PARAM2 are formal
!parameters from the model
!definition being instantiated.
MODEL adder [PA1 = 1, PA2 = 3, PA3 = 4] sum1, sum2 = a1, a2, b1, b2;
I2:half [PARAM1 = PA3, PARAM2 = PA4] sum1 = a1, b1; !PA3 and PA4 are
!passed in from
!the model header.

3.4.4.2 Order-based Actual Parameters
The order-based parameter list contains parameter values, each of which
can be a parameter triple.
The values given in this list are matched with the list of formal
parameters occupying the corresponding locations in the model header.
If you don't want to assign actual values to one or more of the formal
parameters, you can specify the default values with two successive
commas.

Example

[DELAY, , , 10 NS] !2 values are supplied and
!2 values are defaulted

3.4.4.3 Example of Actual Parameter Use

MODEL M1 [PARAM1 = 10 NS, PARAM2 = 20 NS,
PARAM3 = 30 NS, PARAM4 = 40 NS] O = I1, I2;
! 4 formal parameters are defined
EQN [RISE0_1 = PARAM1, FALL1_0 = PARAM2] O = I1 AND SIG; !First gate
EQN [RISE0_1 = PARAM3, FALL1_0 = PARAM4] SIG = NOT I2; !Second gate
ENMODEL M1;
! First instance, order-based actuals are supplied for all 4 formal
!parameters.
! Result: First gate in M1 has RISE0_1 = 50 NS, FALL1_0 = 60 NS.
! Second gate in M1 has RISE0_1 = 70 NS, FALL1_0 = 80 NS.
I1: M1 [50 NS, 60 NS, 70 NS, 80 NS] ... ;
! Second instance, name-based actuals are supplied for all 4 formal
!parameters.
! Result: First gate in M1 has RISE0_1 = 50 NS, FALL1_0 = 60 NS.
! Second gate in M1 has RISE0_1 = 70 NS, FALL1_0 = 80 NS.
I2: M1 [PARAM1=50 NS, PARAM3=70 NS, PARAM2=60 NS, PARAM4=80 NS]...;
! Third instance, no actuals, all 4 formal parameters get their
!default values.
! Result: First gate in M1 has RISE0_1 = 10 NS, FALL1_0 = 20 NS.
! Second gate in M1 has RISE0_1 = 30 NS, FALL1_0 = 40 NS.
I3: M1 ...;
!Fourth instance, order-based, 2 actuals supplied, 2 formal parameters
!get their default values.
!Result: First gate in M1 has RISE0_1 = 50 NS, FALL1_0 = 20 NS.
! Second gate in M1 has RISE0_1 = 70 NS, FALL1_0 = 40 NS.
I4: M1 [50 NS, , 70 NS, ] ... ; !PARAM2 and PARAM4 are defaulted
! Fifth instance, name-based, 2 actuals supplied, 2 formal parameters
!get their default values.
! Result: First gate in M1 has RISE0_1 = 50 NS, FALL1_0 = 20 NS.
! Second gate in M1 has RISE0_1 = 70 NS, FALL1_0 = 40 NS.
I5: M1 [PARAM1=50 NS, PARAM3=70 NS] ... ; !PARAM2 and PARAM4 are
! defaulted.

3-13
3.4.5 Instance Connection Lists

When you instantiate a model, you use its functionality, but you can give its inputs and outputs different names.

The model headers containing the inputs and outputs that you rename (and connect to) come in two forms: signal-name based and pin-name based.

The instance statement allows two ways to match the inputs and outputs: either by pairing the former signal name with a new one (name-based) or by positioning the new instance signal name in the same relative location as the former signal name (order-based).

You can use both order-based and name-based connection lists in the same network, but not within the same instance statement.

3.4.5.1 Name-based Instance Connection Lists

There are two types of name-based lists:

- Those that pair instance-level names with the signal names in any model header.
- Those that pair instance-level names with the pin-names in a pin-name-based model header.

The technique for pairing is the same for both cases:

Write the signal name or pin number used in the model header, followed by a slash (/), and then the new signal name. Since you are pairing each model-level name with an instance-level name, the order of the pairs in the list is not important.

An equal sign is not valid in a name-based list.

Example

```model
MODEL Foo Out = Input1, Input2;                   !signal-based model header
.
.
ENDMODEL Foo;
```

```instance
I1: Foo Input1/a, Out/c, Input2/b;               !instance statement
MODEL Bar 1/Ouch = 2/Nputa, 3/Nputb;            !pin based model header
.
.
ENDMODEL Bar;
```

```instance
I2: Bar 2/a, 1/b, 3/c;                          !instance statement paired
I3: Bar Ouch/a, Nputb/b, Nputa/c;              !instance statement paired
```

Note: If you use a model header name-based signal list and subscript the model header signal names (pin numbers are never subscripted), then you must use the subscripts exactly as they appear in the model definition. A model header signal that is 1-bit wide can be name-based instantiated using signal_name or signal_name<0> in the instance. For example, signals X or X<0> in the instance will match X<0> in the model header.
Example

MODEL X OUT<3:0> = IN ;
.
.
ENDMODEL X;
E1: X OUT<3:0>/A<3:0>, IN/B; !Is OK
E2: X OUT<3,2,1,0>/A<3:0>, IN/B; !Is incorrect
E3: x OUT<3:2>/A<3:2>, OUT<1:0>/A<1:0>, IN/B; ! Is incorrect

3.4.5.2 Order–based Instance Connection Lists

Order–based lists can be used with either signal–based model headers or pin–name–based model headers.

To match the instance–level signal names with the signals in the model header, place the new signal names in the instance statement location corresponding to the location of the signals in the model header. This technique differs from name–based connection lists in that order is important.

Example

MODEL Foo Out = Input1, Input2; !signal–based model header
I1: Foo A = B, C; !order–based instance statement
MODEL Bar 1/Cuch = 2/Nputa, 3/Nputb; !pin–number–based model header
I2: Bar April = May, June; !order–based instance statement

In the above example, I1 instantiates model Foo and calls the signals A, B, and C.

I2 instantiates model Bar and calls the signals April, May, and June.

Order–based lists are most useful when signals are logically interchangeable, as with multi–input gates.

3.4.5.3 Implicit “label.pin” Signal Names

When a label appears on a model instance, a “label.pin” signal name and a “label.pin” signal number are automatically generated for each pin number and formal signal in the model definition. This name joins the label and the pin number with a period, thus:

E4.1
E31.12

These names can be used instead of an explicit name.

The “label.pin” construction is possible whether the connector pins are interpreted as pin–number lists or pin–name lists. See Section 3.3.1.2.2 for a further discussion of pin numbers and pin names.
3.4.5.4 Leaving Signals Unconnected
At instantiation time, signals of a model can be left unconnected. However, an appropriate signal declaration must be provided for those signals. See the OUTPUT and UNCONNECTED_OK declarations. A warning is issued if the proper signal declarations are not used. The method for leaving signals unconnected differs between the two types of instance connection lists.

For an order-based connection list, you enter a null signal as the signal name you wish to leave unconnected (which preserves the ordering of the signals). If the unconnected signal is a primary input, DECSIM generates a new unique name for it, in the form of "$$n\)", where \(n\) is an integer. This name must be used to access the signal with the Interactive Command Language.

Example

```
MODEL M A, B, C = D, E, F;
   ....
   ....
ENDMODEL M;
EI:M X, Y = , Z, ; ! Instance leaving pins
               ! from M unconnected.
```

The signals are connected as follows:

<table>
<thead>
<tr>
<th>Definition Signal</th>
<th>Connected To Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>B</td>
<td>Unconnected</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
</tr>
<tr>
<td>D</td>
<td>Unconnected</td>
</tr>
<tr>
<td>E</td>
<td>Z</td>
</tr>
<tr>
<td>F</td>
<td>Unconnected</td>
</tr>
</tbody>
</table>

For name-based connection lists, signals can be left unconnected either by not mentioning them at all (the number of signals in the instance connection list does not have to equal the number of signals in the definition list), or by including the signal in the connection list but pairing it with a null signal (for example, PIN1/).

Example

```
MODEL M: A, B, C = D, E, F;
   ....
   ....
ENDMODEL M;
M A/X, B/, C/Y, E/Z, F/; ! Instance leaving pins
                          ! from M unconnected.
```
The signals are connected as follows:

<table>
<thead>
<tr>
<th>Definition</th>
<th>Signal</th>
<th>Connected To Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>Unconnected by pairing with null signal</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>Unconnected by not mentioning it</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>Z</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>Unconnected by pairing with null signal</td>
</tr>
</tbody>
</table>

Note: Signals that appear to be unconnected in the model header can have connections if the model has a label. You can make connections to the signals by using `label.pin` and the `SYNONYM` statement.

Example

```
E1: 7400;
E2: 7402 1/E1.2, 2/E1.1, 3/E1.3 ;
```

or

```
E1: 7400;
SYNONYM E1.1 = A, E1.2 = B, E1.3 = C;
E2: 7402 1/B, 2/A, 3/C;
```

3.4.6 Concatenating and Extracting Multibit Signals in Connection Lists

Multibit signals in instance connection lists can connect to other signals of equal width:

```
I1: M1 SIG<7:0> = ... ; !Output connection in this instance
I2: M2 ... = SIG<7:0>; !drives input connection in this instance.
```

It is also possible, however, to connect instances for either of the following reasons:

- To concatenate (or merge) a set of smaller output signals into one larger signal that drives one or more inputs, or
- To extract (or split) smaller input signals from a larger signal that is being driven by an output.

The following sections discuss how to use the network description language to concatenate or merge signals.
3.4.6.1 Concatenating Smaller Signals into a Larger Signal

This section describes the two methods for concatenating (merging) a set of smaller output signals into one larger signal that drives an input. The smaller output signals can be either 1-bit or multibit signals.

The two methods are:

- Make the input name a concatenation of the smaller output names (each of which must be unique). Join the output signal names with the concatenation operator (&). The order of the names (left to right) is MSB→LSB (most significant bit to least significant bit).

- Make the input name be the full subscript range of the output signals. The output signals must have the same name but different subscript ranges. The subscript range has the form <msb:lsb>.

Example illustrating method 1

Network:

```
! Model definitions
MODEL M1 O1 = ... ;
.....
ENDMODEL M1;
MODEL M2 O2<7:0> = ... ;
.....
ENDMODEL M2;
MODEL M3 ... = I3<11:0>;
.....
ENDMODEL M3;

! Instances to connect the models together
! Models with smaller outputs:
I1: M1 O1 / A, ... ;
I2: M2 O2<7:0> / B<7:0>, ... ;
I3: M1 O1 / C, ... ;
I4: M1 O1 / D, ... ;
I5: M1 O1 / E, ... ;

! This model's input is driven by the concatenation of
! the above outputs:
I6: M3 ... , I3<11:0> / A & B<7:0> & C & D & E ;
```
Example illustrating method 2

Network:

```
! Model definitions
MODEL M1 O1 = ...;
......
ENDMODEL M1;
MODEL M2 O2<7:0> = ...;
......
ENDMODEL M2;
MODEL M3 ... = I3<11:0>;
......
ENDMODEL M3;

! Instances to connect the models together
! Models with smaller outputs:
I1: M1 O1 / A<11>, ...;
I2: M2 O2<7:0> / A<10:3>, ...;
I3: M1 O1 / A<2>, ...;
I4: M1 O1 / A<1>, ...;
I5: M1 O1 / A<0>, ...;

! This model's input is driven by the concatenation of
! the above outputs:
I6: M3 ..., I3<11:0> / A<11:0>;
```

3.4.6.2 Extracting Smaller Signals from a Larger Signal

You extract (split) a set of smaller input signals from a larger signal that is driven by an output. The smaller input signals being extracted can be either 1-bit or multibit–wide signals. Also, they can represent overlapping bytes of the driving output.

The output signal has a subscripted signal name of the form: `signal_name< msb : lsb>`. The smaller input signals are named with the appropriate subscript range of the driving output signal.
Example

Network:

! Model definitions
MODEL M1 ... = I1;
......
ENDMODEL M1;
MODEL M2 ... = I2<3:0>;
......
ENDMODEL M2;
MODEL M3 ... = I3<7:0>;
......
ENDMODEL M3;
MODEL M4 O4<11:0> = ... ;
......
ENDMODEL M4;

! Instances to connect the models together
! This model's output drives the following extracted inputs:
I1: M4 O4<11:0> / A<11:0>, ... ;

! Models with smaller inputs:
I2: M1 ... , I1 / A<11>;
I3: M3 ... , I2<7:0> / A<10:3>;
I4: M2 ... , I1 / A<4:3>;
! NOTE: This overlaps "A<10:3>"
I5: M1 ... , I1 / A<1>;
I6: M1 ... , I1 / A<0>;
I7: M1 ... , I1 / A<9>;
! NOTE: this also overlaps "A<10:3>"
3.5 Interconnecting Primitives

With the Interconnect Language, you can connect any DECSIM primitive to any of the other DECSIM primitives: behavior, memory, equation, or MOSFET.

3.5.1 Rules for Interconnecting Primitives

There are two rules that govern the interconnection of primitives by NETPRO:

- Wires can be driven by at most one output.
- Wires must connect to only one type of primitive.

To accommodate wires driven by more than one output, and to accommodate the interconnection of different types of primitives, NETPRO inserts phantom models. The insertion of a phantom because there is more than one output driving a wire is called wire-tying. See Section 3.6.5 for more information.

When a phantom is inserted because there is an interconnection between two different types of primitives, the phantom that is inserted is at the level of the lowest level primitive. However, you can override the default phantom by using the WIRETIE_MODEL_LEVEL declaration. See Section 3.6.5.4 for more information. You can also write your own phantom models. See Section 3.5.3.3 for more information.

3.5.2 Specifying the Type of Phantom

When a primitive is connected to a different type of primitive, there are three types of drivers that can be specified: WIREAND, WIREOR, and TRISTATE. The type of phantom instantiated is determined by a declaration in the model that specifies one of the three phantom types. See Section 3.6.5.1, Section 3.6.5.2, and Section 3.6.5.3 for more information on using these declarations.

Note: The MOS driver phantom models have been set to give a delay of 0.

3.5.3 How DECSIM Models Wire-tying

Wire-tying occurs when two or more outputs are tied together to drive a single signal. This section describes how DECSIM models wire-tying. It also describes the signal declarations associated with wire-tying.

DECSIM modeling of wire-tying handles two problems:

- Outputs that drive high impedance
- Multiple outputs driving conflicting values onto the same signal
3.5.3.1 Overview of the Modeling of Wire-tying
DECSIM models wire-tying differently for each type of primitive (equation, behavior, etc).

Modeling Wire-tying for Equation-level (Gate) Simulation
When NETPRO finds two or more equation primitive outputs tied together, it restructures the network by inserting an equation-level model called a phantom. You can override the default phantom model either by writing your own phantom model (see Section 3.5.3.3) or by specifying that another level of model be used (see Section 3.6.5.4). See the “Insertion of Phantom Models” section below for more information.

Modeling Wire-tying for MOS-level Simulation
MOS modeling does not have outputs that are tied together; it only has pullup and pulldown paths, attached to nodes. The MOS algorithm handles the evaluation of multiple pullup and pulldown paths for a node by computing the node voltage due to the pullup/pulldown voltage divider, then calculating which state value (0, 1, U) that voltage represents.

Modeling Wire-tying for Behavior-level Simulation
When NETPRO finds two or more Behavior ports tied together, it restructures the network by inserting a Behavior-level model called a phantom. You can override the default phantom model either by writing your own phantom model (see Section 3.5.3.3) or by specifying that another level of model be used (see Section 3.6.5.4). See the “Insertion of Phantom Models” section below for more information.

3.5.3.2 Insertion of Phantom Models into Your Network
This section discusses only phantom models as they pertain to wire-tyied equation primitive outputs.

When NETPRO discovers multiple outputs tied together, it restructures the network as follows:
You use wire–tie declarations to specify your technology and control which phantom is inserted in your network. See Section 3.6.5 for more information.

The phantom is a model that has been given a special name and is already defined in a library. Standard phantom models are defined in the GATES.NET library.

The special name allows NETPRO to find the correct phantom model to insert. The model name has different fields that indicate the type of phantom, how many inputs it has, etc.

Different phantoms are created by taking one name from each of five fields and concatenating all five names into a single name. Whether a phantom is TRISTATE, WIREAND, PULLUP, etc. is determined by the wire–tie declaration given for the signal.

To generate a phantom name, concatenate the following fields:

<table>
<thead>
<tr>
<th>Field 1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
<th>Field 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SCALAR_</td>
<td>TRISTATE_</td>
<td>BLANK_</td>
<td>PHANTOM_</td>
<td>number_of_inputs</td>
</tr>
<tr>
<td>WIREAND_</td>
<td>PULLUP_</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WIREOR_</td>
<td>PULLDOWN_</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WIREMOS_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For example, the model name for a TRISTATE with 14 inputs is:

%SCALAR_TRISTATE_PLAIN_PHANTOM_14

3–23
If you use a PULLDOWN declaration in conjunction with the signals, your model name might look this:

%SCALAR_TRISTATE_PULLDOWN_PHANTOM_14

When phantom models are inserted in your network, they are given implicit labels in the form $PHANTOM_signal_name. The signal name used is the output of the phantom. If this signal is in the label.pin form, the phantom label is $PHANTOM_signalname_label$pin. If the phantom is multibit, the label is $PHANTOM_signal_name<31:0>$. These labels are visible from the DECSIM Command Language.

To minimize the total number of phantom models and to give every phantom model access to the original contributing outputs, the insertion of a phantom is deferred until it is not possible to attach another wire–tied output. Therefore, when a signal connects to wire–tied outputs and runs through several levels of hierarchy, NETPRO inserts only one phantom that connects to all the outputs from the different levels. NETPRO does not insert a different phantom at each level.

Note: WIREMOS phantoms are not supported in ZYCAD mode.

3.5.3.3 User–defined Phantoms

If the phantoms provided do not meet your needs, DECSIM allows you to define your own. By replacing the definition for one or more of the above technologies, you can redefine the appropriate keywords.

Restrictions for User–defined Phantoms

DECSIM allows more than one model definition to exist for any one model name. If you redefine a phantom, you create a second definition for the same name. It is helpful to know how DECSIM decides which model to instantiate. (For a list of rules, see Section 3.2.2).

Note: Phantom redefinition is not supported in the ZYCAD environment.

Phantom models should be written with equations having no more than 4 inputs each. However, the most efficient model is the one with the fewest equations, so it is possible that special circumstances demanding the greatest possible efficiency may lead you to define a phantom with 5 or 6 inputs on one of its equations.

To avoid another implementation restriction, DECSIM inserts %SCALAR_TRISTATE.PLAIN_PHANTOM_1 in the network wherever it is needed to eliminate zero–length feedback. That is, whenever a model instance ties the output of an equation directly to its own input, the above–named phantom is inserted in the loop. Zero–length feedback is extremely rare in actual networks. However, the user who redefines the %SCALAR_TRISTATE.PLAIN_PHANTOM_1 should be aware of this possible usage.
Equations Suitable for Phantoms and Phantom Contributors

Appendix D lists several equation operators that are suitable for defining phantom models and phantom–contributors (models that normally drive the inputs of phantoms). This section emphasizes these operators and explains their use.

The operators CONV1Z, CONV0Z, TS, and MOSTS are especially useful for writing phantom–contributer models. CONV1Z (any expression) converts an ordinary equation into an open–collector model. Likewise, CONV0Z converts an ordinary equation to an open–emitter model. The TS operator models the enable input of a tristate driver. MOSTS models a unidirectional MOS transmission gate. MOSTS differs from TS only in that a Z data input is transmitted as a Z output in MOSTS and as a U in TS.

Note: The CONVxx operator is low in precedence. Operations in the expression are performed first, then the entire expression is converted.

A special operator, TSCON (tristate consensus), exists for constructing the tristate phantom equation.

Example

EQN out = in1 TSCON in2 TSCON in3 TSCON in4;

The operators CONVZ1 and CONVZ0 model the PULLUP and PULLDOWN features.

Example (taken from the TRISTATE_PULLUP model)

EQN out = CONVZ1 (in1 TSCON in2 TSCON in3);

There are no operators specifically for wireand–consensus or wireor–consensus. The AND and OR operators must be used. These operators normally interpret a Z input as if it were a U, so an equation such as the one below is required. This example is taken from the WIREAND_PULLUP model:

EQN out = (CONVZ1 in1) AND (CONVZ1 in2) AND (CONVZ1 in3);

3.6 Signal Declarations

3.6.1 Definition of Signal Declarations

A signal declaration defines a property of the signals named in its associated list. Signal declarations suppress error checking or supply a value to be associated with a signal. You can place a signal declaration anywhere in a subblock.
Interconnect Language

Format

Signal_declaration signal_name [ , ... ];

Example

IN_STUB DATAIN_1, DATAIN_2, E1.2;
OUT_STUB DATAOUT_3, DATAOUT_4;

The format varies with some signal declarations. See specific declarations for more detailed information.

3.6.2 Model Header Signal Declarations

Model header signal declarations affect only the signals in the model header. They can appear within a structure subblock or within the GLOBAL–ENDGLOBAL block.

Signal declarations placed in the GLOBAL–ENDGLOBAL block have scope over all subsequent structure subblocks in the model definition. In other words, placing a signal declaration in the GLOBAL–ENDGLOBAL block is equivalent to placing a duplicate of that declaration in every structure subblock within the model definition.

3.6.2.1 BIDIRECTIONAL Declaration

BIDIRECTIONAL is a model header signal declaration that converts a model header signal to a bidirectional signal. There are two cases where you use the BIDIRECTIONAL declaration.

- A signal connects to both inputs and outputs inside the model.
- A signal connects to one or more MOSFET source or drains inside the model.

These cases are handled differently. For example, in the following illustration, a signal connects to both inputs and outputs inside the model:

![Diagram showing a signal connecting to both inputs and outputs inside the model]

In this case, there are three ways you can declare the model header signal.

Note: If you do not use one of these three ways, NETPRO issues compile warnings.

- You can put the signal on the left (output) side of the equal sign in the model header and not supply a BIDIRECTIONAL declaration for the signal. In this case, NETPRO does not issue any warnings, but interprets the signal as an output.
You can put the signal on the left (output) side of the equal sign in the model header and explicitly supply a BIDIRECTIONAL declaration for it. In this case, a wire-tying declaration (like TRISTATE, etc.) must accompany the BIDIRECTIONAL declaration.

You can put the signal on the right (input) side of the equal sign in the model header and explicitly supply a BIDIRECTIONAL declaration for it. In this case, a wire-tying declaration (like TRISTATE, etc.) must accompany the BIDIRECTIONAL declaration.

The following illustration shows the second case where you use the BIDIRECTIONAL declaration. A signal connects to one or more MOSFET source or drains inside the model.

In this case, you can put the signal on either side of the equal sign in the model header. The BIDIRECTIONAL declaration is optional.

**Format**

```plaintext
BIDIRECTIONAL signal name1 [ , ... ];
```

**Example**

```plaintext
BIDIRECTIONAL XCVR1, BUS_A, E3.14;
```

---

### 3.6.2.2 IN_STUB and OUT_STUB Declarations

IN_STUB and OUT_STUB identify a model header signal that has no connections to it inside the model. This normally happens during model development, before the model is complete.

A signal name associated with IN_STUB is not accessible from the DECSIM command language. But you can access this signal by the name given to it in the model network.

If your model has no real driver for a model header output, then the OUT_STUB declaration causes DECSIM to insert a primary input as a driver.

The OUT_STUB declaration can take an assignment of a state value 0, 1, U, or Z, allowing you to initialize a signal to a specified value. The primary input receives the initialization values when the network is compiled and loaded into DECSIM.
Format

IN_STUB signal, signal, ...
OUT_STUB signal = init_value, signal = init_value, ...

Example

OUT_STUB A = 1;
OUT_STUB A, B = 0, C;
OUT_STUB A<7:0> = 10101010#2;  ! Simple 1-bit value
OUT_STUB A<7:0> = U10U10#2;    ! A and C do not have
OUT_STUB A<7:0> = 1A#16;       ! an initialization value
OUT_STUB A<7:0> = 11111111#2;  ! Radix 2 value
OUT_STUB A<7:0> = 11110110#2;  ! Radix 2 with U's and Z's
OUT_STUB A<7:0> = 1A#16;       ! Radix 16
OUT_STUB A<7:0> = 1111111111#2; ! Too wide; causes warning

Initialization values are optional. If not specified, the value is undefined.
Both signals and initialization values can be multibit. The value can be
specified in any radix, and can contain all four logic states. If the
initialization value width is less than the signal width, it is zero-filled
to the most significant bit. If it is greater than the signal width, it is
truncated and a warning is issued.

Note that if you want to assign the same value (for example, 0) to signals
A and B, you must explicitly state OUT_STUB A=0, B=0. You cannot
simply say OUT_STUB A,B=0.

3.6.2.3 UNCONNECTED_OK Declaration

The UNCONNECTED_OK declaration identifies model–header signals
that can be intentionally left unconnected when the model is instantiated.
If signals in this declaration are left unconnected, there is no warning
message. If the signals are connected, this declaration is ignored.

Note: Error checking for this declaration does not occur until you try to
instantiate your model.

Example

UNCONNECTED_OK CLEAR, IN1, OUT1;

If you use a signal declared to be unconnected as a primary input to
another model, DECSIM generates a new unique name for the signal in
the form of $$n, where n is an integer. This name, not the original name,
must be used to access the signal.

The UNCONNECTED_OK declaration can also take an assignment value
on an input signal, allowing you to initialize a signal to a specified
value. The signal receives the initialization value when the network is
compiled and loaded into DECSIM. Trying to assign a value to an output
or bidirectional signal results in a warning.
Format

UNCONNECTED_OK signal = init_value, signal = init_value, ...;

Example

UNCONNECTED_OK A = 1; ! Simple 1-bit value
UNCONNECTED_OK A, B = 0, C; ! A and C do do not have
UNCONNECTED_OK A<7:0> = 10101010#2; ! an initialization value
UNCONNECTED_OK A<7:0> = 1101010#2; ! Radix 2 value
UNCONNECTED_OK A<7:0> = 01110111#2; ! Radix 2 with U's and Z's
UNCONNECTED_OK A<7:0> = 1A#16; ! Radix 16
UNCONNECTED_OK A<7:0> = 11111111#2; ! Too wide; causes warning

Initialization values are optional. If not specified, the value is undefined.
Both signals and initialization values can be multibit. The value can
be specified in any radix, and can contain all four logic states. If the
initialization value width is less than the signal width, it is zero–filled
to the most significant bit. If it is greater than the signal width, it is
truncated and a warning is issued.

Note that if you want to assign the same value (for example, 0) to signals
A and B, you must explicitly state UNCONNECTED_OK A=0, B=0; You
cannot simply say UNCONNECTED_OK A,B=0;

3.6.3 INPUT and OUTPUT Declarations

The INPUT declaration identifies a nonheader, or internal, signal that has
no output driving it. The OUTPUT declaration identifies a nonheader, or
internal, signal that drives no inputs.

The declarations indicate that you have intentionally left the signals
unconnected and turn off the warning message.

INPUT and OUTPUT cannot be specified for model header signals.

INPUT declarations can take assignments of values, allowing you to
initialize a primary input (undriven input signal) to a specified value.
The primary input receives the initialization value when the network is
compiled and loaded into DECSIM.

Format

INPUT signal = init_value, signal = init_value, ...;
OUTPUT signal,...signal;

Example

INPUT A = 1; ! Simple 1-bit value
INPUT A, B = 0, C; ! A and C do do not have
INPUT A<7:0> = 10101010#2; ! an initialization value
INPUT A<7:0> = UZ10U10#2; ! Radix 2 value
INPUT A<7:0> = UZ10U10#2; ! Radix 2 with U’s and Z’s
INPUT A<7:0> = 1A#16; ! Radix 16
INPUT A<7:0> = 11111111#2; ! Too wide; causes warning

3–29
Initialization values are optional. If not specified, the value is undefined. Both signals and initialization values can be multibit. The value can be specified in any radix, and can contain all four logic states. If the initialization value width is less than the signal width, it is zero-filled to the most significant bit. If it is greater than the signal width, it is truncated and a warning is issued.

Note that if you want to assign the same value (for example, 0) to signals a and b, you must explicitly state INPUT A=0, B=0. You cannot simply say INPUT A,B=0;

Only primary inputs (undriven input signals) can be initialized; you cannot initialize MOS nodes or primitive outputs.

3.6.4 SYNONYM Declaration

SYNONYM assigns alternate names to any signal. All names are accessible from the interactive command language.

Declaring signals to be synonyms of each other connects them. Signals declared synonymous must be defined in the same model. A synonym in one model cannot be created for a signal defined in another model.

Format

SYNONYM signal_namela = signal_namemb [ ,list of name pairs ];

Example

SYNONYM e4.12 = ovfl h; !label.pin naming convention, not hierarchy
SYNONYM primary_out1 = primary_in1; !"feedthrough" wire

The DECSIM command language accepts all synonym names, but prints only the primary name. The header signal name is always the primary name. Otherwise, the primary name is to the left of the equal sign in the last SYNONYM statement. All other names are considered secondary names. Label.pin is never the primary name.

Example

SYNONYM A = B; !A is the primary name for this signal
SYNONYM B = C; !B is now the primary name
SYNONYM D = A; !D is now the primary name
SYNONYM E3.5 = B; !D is still the primary name

When you use SYNONYM to connect two or more model header signals, NETPRO makes some assumptions and supplies some declarations for you.

Signals in a model header that are declared SYNONYM can be connected to internal signals. The following table shows which default declarations are supplied by NETPRO and which declarations you must specify for a model header such as in the example below:
Example

Model sample out1,out2 = in1,in2;

Table 3-1  Declarations with Header Signal SYNONYM

<table>
<thead>
<tr>
<th>Internal Signals Connected to the Header Signal</th>
<th>SYN out1 = out2;</th>
<th>SYN in1 = in2;</th>
<th>SYN out1 = in1;</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Default: OUTPUT. You must specify OUT_STUB</td>
<td>Default: INPUT. You must specify IN_STUB</td>
<td>Default: INPUT</td>
</tr>
<tr>
<td>inputs</td>
<td>Default: BIDIRECTIONAL. You must specify OUT_STUB</td>
<td>Default: INPUT</td>
<td>Default: INPUT</td>
</tr>
<tr>
<td>outputs</td>
<td>Default: OUTPUT BIDIRECTIONAL. You must specify IN_STUB</td>
<td>Default: BIDIRECTIONAL</td>
<td></td>
</tr>
<tr>
<td>inputs and outputs, or bidirectional signals</td>
<td>Default: BIDIRECTIONAL</td>
<td>Default: BIDIRECTIONAL</td>
<td>Default: BIDIRECTIONAL</td>
</tr>
</tbody>
</table>

Subscripted multibit signal synonyms are allowed as long as the total number of bits represented by each synonym argument is the same.

Example

SYNONYM a<7:0> = b<15:0>;

Note: SYNONYM x <31:1> = x <30:0>; wire-ties all bits of x. The primary name is x <31>. SYNONYM x <0:30> = x <1:31>; performs the same function, but the primary name is x <0>.

3.6.5  Signal Declarations for Wire-tying

You use wire-tying declarations whenever two or more outputs are tied together (or might be tied together later), or when BIDIRECTIONAL is declared on a model header signal that connects to both an output and an input.

The type of wire-tie declaration you use depends on the technology of your model. Wire-tie declarations can be used for model header signals and for internal signals inside the model. All wire-tie declarations have similar syntax.

Format

```
Wire_tie_declaration signal-name [ , ... ];
```

There are four declarations that determine the type of phantom model that NETPRO inserts into your network: WIREAND, WIREOR, TRISTATE, and WIREMOS. There is also a declaration that controls the level at which a phantom is modeled; it is called the WIRETIE_MODEL_LEVEL declaration (see Section 3.6.5.4).
3.6.5.1 WIREAND Signal Declaration
The WIREAND declaration specifies the insertion of a phantom to model an open-collector TTL gate. Each driver is expected to output either 0, Z, or U.

Format

WIREAND signal-name [ , ... ];

Example

WIREAND AREG, BREG, CREG;

Below is a sample phantom model as defined in the DECSIM-supplied library. DECSIM inserts this phantom model into your network when you use WIREAND for three signals.

MODEL %SCALAR_WIREAND.PLAIN_PHANTOM_3 OUT = IN1, IN2, IN3;
STRUCTURE;
  EQN OUT = CONVZ ((CONVZ1 IN1) AND (CONVZ1 IN2) AND (CONVZ1 IN3));
ENDSTRUCTURE;
ENDMODEL %SCALAR_WIREAND.PLAIN_PHANTOM_3;

The truth table below shows the result of using WIREAND for a 2-input WIREAND phantom.

<table>
<thead>
<tr>
<th>Input1</th>
<th>Input2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>0</td>
</tr>
</tbody>
</table>

You can model a passive pullup resistor with a PULLUP declaration. The truth table for such a resistor is the same as the one above, except that 1 replaces Z in the matrix.

Normally, you would use WIREAND in conjunction with PULLUP. NETPRO issues a warning message if it isn't. NETPRO also issues a warning message if the driver of the WIREAND phantom drives an invalid state (a 1).

3.6.5.2 WIREOR Signal Declaration
The WIREOR declaration specifies the insertion of a phantom to model ECL or open-emitter gates. Each driver is expected to output either 1, Z, or U.
Format

```plaintext
WIREOR signal-name [ , ... ];
```

Example

```plaintext
WIREOR AREG, BREG;
```

Below is a sample phantom model with two inputs as it is inserted in your network.

```plaintext
MODEL %SCALAR_WIREOR_plain_phantom_2 OUT = IN1, IN2;
STRUCTURE;
  EQN OUT = CONVZ0 ((CONVZ0 IN1) OR (CONVZ0 IN2));
ENDSTRUCTURE;
ENDMODEL %SCALAR_WIREOR_plain_phantom_2;
```

The truth table below shows the result of using WIREOR for a 2-input WIREOR phantom.

<table>
<thead>
<tr>
<th>Input1</th>
<th>Input2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Z</td>
</tr>
<tr>
<td>0</td>
<td>Z</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>Z</td>
<td>1</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>1</td>
</tr>
</tbody>
</table>

You can model a terminator resistor (passive pulldown to ground) with a PULLDOWN declaration. The truth table for such a resistor is the same as the one above except that 0 replaces Z in the matrix.

Normally, you would use WIREOR in conjunction with PULLDOWN. NETPRO issues a warning message if it isn't. NETPRO also issues a warning message if the driver of the WIREOR phantom drives an invalid state (a 0).

### 3.6.5.3 TRISTATE, WIREMOS Signal Declarations

The TRISTATE declaration specifies the insertion of a phantom to model TTL Tristate technology. Drivers can output all four states (0, 1, Z, and U).

Format

```plaintext
TRISTATE signal-name [ , ... ];
```
Example

TRISTATE A1,A2,C3;

Below is a sample TRISTATE phantom model as it is inserted in your network.

MODEL %SCALAR_TRISTATE_PLAIN_PHANTOM_3 OUT = IN1,IN2,IN3;
STRUCTURE;
EQN OUT = IN1 TSCON IN2 TSCON IN3;
ENDSTRUCTURE;
ENDMODEL %SCALAR_TRISTATE_PLAIN_PHANTOM_3;

TRISTATE is the default wire-tying declaration.

The WIREMOS declaration models MOS transistor logic, using gate-level models.

Format

WIREMOS signal name [ , ... ];

The truth table below shows the results of using either TRISTATE or WIREMOS.

<table>
<thead>
<tr>
<th>Table 3–4 Truth for TRISTATE or WIREMOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input1</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>Z</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

NETPRO issues a warning message if the driver of the TRISTATE or WIREMOS phantom cannot drive the Z#2 state.

3.6.5.4 WIRETIE_MODEL_LEVEL Signal Declaration

Controls the level of detail at which the phantom is modeled. The primary purpose of the WIRETIE_MODEL_LEVEL declaration is to allow you to apply stimuli with DEPOSIT/FORCE to multibit signals in Behavior. In order to do this, you must declare the signal to be wiretied at the gate level.

Format

WIRETIE_MODEL_LEVEL signal = model_level

where model_level can be one of three keywords: MOS, GATE, or VECTOR

By default, NETPRO models a phantom equivalent to the lowest level model that drives the phantom: VECTOR (Behavior) is high; GATE (Scalar) is intermediate, and MOS is low. However, there are cases where the wiretying should be done at a lower level than the default level. The example below shows a signal that is driven by two Behavioral models, but the phantom will be modeled at the gate level.
Example

\texttt{WIRETIE\_MODEL\_LEVEL x\langle 31:0 \rangle = GATE}

There are also cases where you may want to write your own phantom. See Section 3.5.3.3 for more information on user-defined phantoms.

\textbf{Note:} You cannot specify a phantom model that is modeled at a higher level than the default phantom model.

### 3.6.6 TIMEOUT Declaration in Equation Primitives

The \texttt{TIMEOUT} declaration allows you to specify a decay time for a signal in an equation primitive phantom. The time delay specified in the \texttt{TIMEOUT} declaration is passed as a parameter to the \texttt{WIREMOS} phantom model, which contains the formal parameter “timeout”. The \texttt{TIMEOUT} declaration, in conjunction with phantoms, allows you to model the charge storage of MOS nodes, even though you are modeling with equation primitives.

\textbf{Format}

\texttt{TIMEOUT signame = \textbf{time} [ , signame = \textbf{time}, ... ];}

See Section 3.10.1 for time units. The default time unit is nS (nanoseconds).

The following example of a phantom is taken from the GATES.NET library file.

\textbf{Example}

\texttt{MODEL \%SCALAR\_WIREMOS\_PLAIN\_PHANTOM\_2 [TIMEOUT=0] OUT = IN1, IN2;}
\texttt{STRUCTURE;}
\texttt{EQN [RISE0 \_Z=TIMEOUT, FALL1 \_Z=TIMEOUT,}
\texttt{RISE0 \_I=0, FALL1 \_I=0,RISEZ \_I=0,FALLZ \_I=0]}
\texttt{OUT = IN1 TCON IN2;}
\texttt{ENDSTRUCTURE;}
\texttt{ENDMODEL \%SCALAR\_WIREMOS\_PLAIN\_PHANTOM\_2;}

If there is no \texttt{TIMEOUT} declaration, the delay in the phantom is set to the formal default value in the phantom definition.

If you specify \texttt{TIMEOUT}, and no phantom is inserted for the signal, the declaration is ignored with no warning.

If you specify \texttt{TIMEOUT}, and a phantom without a “timeout” formal parameter is inserted for the signal, a warning is issued.

\texttt{TIMEOUT} can be set to a time value, but not to the keyword \texttt{INFINITY}.

The \texttt{TIMEOUT} value can only be passed into a phantom containing the formal parameter “timeout”. The only phantom in GATES.NET containing that formal parameter is the \texttt{WIREMOS} phantom.

The \texttt{DEFAULT\_TIMEOUT} statement at the top level and the \texttt{DEFAULT\_TIMEOUT} statement in the process parameter file affect only MOS nodes. They do not affect the \texttt{TIMEOUT} value for equation primitive phantoms.
Interconnect Language

3.6.6.1 NO_WIRETIE_CHECK Signal Declaration
The NO_WIRETIE_CHECK declaration is used with the TRISTATE, WIREOR, WIREMOS, or WIREAND declarations to disable checking by NETPRO. NO_WIRETIE_CHECK declared alone is interpreted by NETPRO as if it were used with TRISTATE.

WIRETIE is an alternate keyword that is equivalent to NO_WIRETIE_CHECK.

The following checks are disabled by NO_WIRETIE:

- All equations driving the signal must be able to drive the high impedance (Z#2) state.
- If the signal is declared WIREAND, no equation driving the signal can have the capability of driving the 1#2 state.
- If the signal is declared WIREOR, no equation driving the signal can have the capability of driving the 0#2 state.
- If the signal is declared WIREAND, a PULLUP declaration must also be supplied.
- If the signal is declared WIREOR, a PULLDOWN declaration must also be declared.

3.6.6.2 PULLUP, PULLDOWN Signal Declarations
The PULLUP and PULLDOWN declarations are used alone or with one of the above wire-tying declarations to model a pullup or pulldown resistor. They cause the phantom model to be either PULLUP or PULLDOWN.

Format

\[
\begin{align*}
\text{PULLUP} & \quad \text{signal-name} [ , \ldots ]; \\
\text{PULLDOWN} & \quad \text{signal-name} [ , \ldots ];
\end{align*}
\]

A high-impedance (Z) state is “pulled up” to 1 if the affected signal is listed in a PULLUP declaration for open-collector, open-MOS, and tristate elements, and “pulled down” to 0 if the signal is listed in a PULLDOWN declaration for open-emitter, open-MOS, and tristate elements. An open-MOS or tristate output can be listed in one or the other declaration, but not both. As the name implies, a PULLUP or PULLDOWN declaration is analogous to a pullup or pulldown resistor in a hardware network.

If the structure or behavior model definition contains a PULLUP or PULLDOWN declaration, a separate declaration in the network description is not required.

If you use a signal declared to be a PULLUP or PULLDOWN as a primary input to another model, DECSIM generates a new unique name for the signal in the form $$\$n$$, where n is an integer. This name, not the original name, must be used to access the signal from the Interactive Command Language.
3.6.6.3 Consistency Checks for Wire-tying Declarations

NETPRO checks that signals declared WIREAND, WIREOR, TRISTATE, and WIREMOS are used with the correct technology (open-emitter, open-collector, etc.). Any equation driving these signal types must allow some input combination to cause a Z output. In addition, WIREAND equations must never output a 1. A WIREAND signal must have a PULLUP declared, or else the signal is undefined half the time. Likewise, WIREOR equations must never output a 0. A WIREOR signal must have a PULLDOWN declared, or else the signal is undefined half the time. NETPRO issues a compile-time warning if these rules are broken.

Because of the definition of the technologies, the WIREAND_PLAIN, WIREAND_PULLDOWN, WIREOR_PLAIN, and WIREOR_PULLUP phantoms do not have applications. These phantoms are included only to model what happens if your network description has the indicated combination of declarations, and you choose to ignore DECSIM's warning messages. You can redefine these four phantoms to achieve special effects; however, the warning messages from NETPRO are not silenced.

3.6.7 MOS Declarations

There are two types of MOS declarations: the node attribute declarations and the wiretying declarations. The node attribute declarations are described below; the wiretying declarations, which are WIREAND, WIREOR, and TRISTATE, are described in Section 3.6.5.

3.6.7.1 Specifying Node Attributes

In DECSIM MOS, each node in the circuit has the following properties:

- Unique name
- High and low thresholds
- Timeout value
- Charge-sharing (enabled or disabled)
- Node lumped capacitance (sum of parasitic capacitance, gate capacitance, and fixed node capacitance)
- Optional fixed node delay (to override dynamically calculated delay)

The network description language contains various declarations that control these attributes for individual nodes.

3.6.7.2 POWER Declaration

For NETPRO to detect pullup and pulldown paths, you must identify the signals that are connected to power supply or ground.
Interconnect Language

Format

POWER signame = value [, signame = value, ...];
where value is a logic state (0 or 1).

Example

POWER VDD=1, VSS=0;

Resolving Multiple Declarations at the Same Level

A signal can not be declared both ‘POWER signame = 1;’ and ‘POWER signame = 0;’. A warning message is issued, and one of these is arbitrarily discarded.

Resolving Multiple Declarations at Different Levels

POWER can be declared on a model’s header signal. If a lower–level model header signal is declared power, then at the current (higher) level, that signal is also power. You can specify identical power declarations at different levels without any problems. However, if a power declaration of a different value is supplied, a warning message is issued.

3.6.7.3 CAPACITANCE Declaration

The CAPACITANCE declaration assigns a fixed capacitance value to a node. The default units for capacitance are PF (picofarads).

Format

CAPACITANCE signame = capacitance [, signame = capacitance, ...];
where capacitance is as specified in the UNITS Chapter.

Example

CAPACITANCE SIG1 = 9;          !Defaults to 9 PF (picofarads)
CAPACITANCE SIG1 = 8 PF;        !Units specified explicitly
CAPACITANCE SIG1<7:0> = 7 PF;   !All eight bits of SIG1 get the value

The capacitance declaration specifies capacitance for a node to be added to the capacitance calculated from the area of the transistor gates, sources, and drains. Typically, the additional capacitance represents the capacitance of the interconnect. See Section 3.6.7.4 and Section 3.6.7.5 for details on how to override the source and drain capacitance summing.

If a CAPACITANCE statement is not supplied for a node, NETPRO adds a fixed capacitance value to the node capacitance by default. This value is specified by the NODECAPAC_DEFAULT parameter in the process parameter file. If a CAPACITANCE statement is supplied for a node, this NODECAPAC_DEFAULT value is not added to the node capacitance.
Resolving Multiple Declarations at the Same Level

More than one CAPACITANCE declaration is permitted for a single signal. The capacitance values are summed.

Resolving Multiple Declarations at Different Levels

If a model header signal has a CAPACITANCE declaration, the capacitance value is carried up into any higher-level model that instantiates the model, where it is added to any capacitance values at the higher level.

Placing the CAPACITANCE Declaration on an Internal Node in a Stack

The capacitance value is added into the capacitance sum used by the node at the "top" of the stack.

3.6.7.4 PARASITIC Declaration

Normally, DECSIM calculates node capacitance by summing: (1) the capacitances of all sources, drains, and gates of connected transistors; and (2) the capacitances specified by any CAPACITANCE statements for that node. However, the PARASITIC statement allows you to specify a capacitance value that replaces the capacitance contributed by the connected sources and drains, although the gate capacitance contribution is still used. You still do not supply a capacitance value directly, but instead supply the area and periphery values from which NETPRO calculates the capacitance.

Format

PARASITIC signame = [(area, periphery, parasitic-type), ... ], ... ;

Where:

- area is an area as specified in the UNITS Chapter.
- periphery is a distance as specified in the UNITS Chapter.
- parasitic-type is a string that should match a parasitic-parameter-set defined in the parasitic_parameter section of the Process Parameter File. This set specifies the appropriate coefficients for converting the area and periphery into a capacitance value. The string is not enclosed in quotation marks.

The default parasitic-type is specified in the Process Parameter File and the default area and periphery are calculated from values specified in that file.
Resolving Multiple Declarations at the Same Level

Multiple PARASITIC declarations for a single node are summed.

Resolving Multiple Declarations at Different Levels

If "signame" is a model header signal, the model is instantiated in another network and signal "signame" is connected to other transistors. In this event, the PARASITIC statement in the original (lower–level) model does not inhibit capacitance calculations for the "other" transistors in models outside the original. This is because PARASITIC is a replacement capacitance value that can be calculated only if you know the configuration of the circuits. It is impossible for you to predict what the capacitance should be for the next level without complete knowledge of the circuit layout.

Placing This Declaration on an Intermediate Node in a Stack

Placing PARASITIC on an intermediate node replaces the capacitance of the connecting source and drain of the two transistors attached to the node. The capacitance sum is added to the connecting node at the "top" of the stack (if the CHAIN_INTER_CONTRIB switch in the Process Parameter file is set to TRUE).

3.6.7.5 PCAP Declaration

The PCAP declaration, a special form of the PARASITIC declaration, is equivalent to a PARASITIC statement whose parasitic-type is the string "PCAP".

Format

PCAP signame = [area, periphery] [, ...];

Example

PCAP SIG1 = (100, 10); !Defaults to 100 SQ UM (sq microns), 110 microns
PCAP SIG1 = (100 SQ UM, 10 UM); !Units specified explicitly
PCAP SIG1<7:0> = [100, 10]; !All eight bits of SIG1 get the value

Note: To use the PCAP declaration, a parasitic_parameter section named "PCAP" must be in the Process Parameter File.

3.6.7.6 THRESHOLD Declaration

The THRESHOLD declaration assigns the logic thresholds for nodes. If a node threshold is not assigned explicitly, the default for the given process is assumed (as specified in the Process Parameter File). Thresholds are specified as a fraction of Vdd and thus are always a dimensionless quantity in the range 0.0 to 1.0. Thresholds greater than 1 or less than 0 are ignored.
Interconnect Language

Format

!Order-based
THRESHOLD signame = [highthresh, lowthresh];
[,...signame = [highthresh, lowthresh], ...];

!Name-based
THRESHOLD signame = [HIGH=highthresh, LOW=lowthresh];
[,...signame = [HIGH=highthresh, LOW=lowthresh], ...];

Example

THRESHOLD SIG1 = [.8, .2];
THRESHOLD SIG1 = [HIGH=0.8, LOW=0.2];
THRESHOLD SIG1<7:0> = [.8, .2];

Placing a THRESHOLD statement on a signal forces that signal to be modeled as a node (never modeled as internal to a stack).

Resolving Multiple Declarations at the Same Level

The DECSIM MOS simulator assumes that all transistors connected to the same node have the same thresholds. Therefore, multiple conflicting thresholds specified for one node yield a compile-time warning, and NETPRO selects the maximum of the high and the minimum of the low thresholds. Thresholds are always specified in pairs; if one is omitted, the single value specified is used for both (with a warning message). The high threshold must be greater than or equal to the low threshold. If this rule is violated, the THRESHOLD statement is ignored, and a warning message is issued.

Resolving Multiple Declarations at Different Levels

THRESHOLD can be declared for a model's header signals. A THRESHOLD statement in a higher-level model overrides the thresholds specified by any model connected to a given signal. A warning is issued when two conflicting THRESHOLD statements are placed on two separate models' header pins that are connected together at the same level with no overriding THRESHOLD statement.

Note: The voltage thresholds predict the logic state of a node, given the voltage to which the node is being driven. The thresholds have no effect on the calculated delay.

3.6.7.7 DELAY Declaration

The DELAY declaration specifies a fixed delay value for a node and overrides the usual dynamically calculated delay.
Interconnect Language

Format

!Order-based
DELAY signage = [triple, ...] [,signage = [triple, ...], ...];

!Name-based
DELAY signage = [paramkeyword = triple, paramkeyword = triple [, ...]]; [,signage = [paramkeyword = triple, paramkeyword = triple [, ...]]];

Example

DELAY SIG1 = [20, 10]; !Defaults to NS (nanoseconds)
DELAY SIG1 = [20 NS, 10 NS]; !Units are specified explicitly
DELAY SIG1 = [RISE0, 1=20, FALL1, 0=10]; !Name-based
DELAY SIG1[7:0] = [20, 10]; !All eight bits of SIG
!get the value

Placing a DELAY statement on a signal forces that signal to be modeled as a node (never modeled as internal to a stack).

The syntax of MOS node delay is identical to equation delay parameters, except that:

- "Z" delays are ignored.
- Parameter passing is not permitted; only explicit times are accepted.

Time multipliers allowed are S, MS, US, NS, PS, FS. Normal time defaults apply. If all the rise/fall parameters are not supplied for the node, NETPRO defaults the missing parameters exactly as it does for EQN statements.

Resolving Multiple Declarations at the Same Level

DELAY for a given signal can be declared only once; a warning message is issued otherwise, and one of the delay sets is arbitrarily thrown out.

Resolving Multiple Declarations at Different Levels

DELAY is allowed for a model header signal. A DELAY declaration in higher-level model overrides delays declared inside the model. A warning is given if header signals (from different models) have conflicting delays and are connected to each other; one delay set is selected arbitrarily and the rest are thrown out. To correct the warning, specify a delay in the next level of the hierarchy.
3.6.7.8 NO_CHARGE_SHARE Declaration

The NO_CHARGE_SHARE declaration disables part of the charge sharing calculation during simulation. DECSIM invokes a charge sharing computation during simulation when a transmission gate starts to conduct. If a node state is predicted to change state due to charge sharing, and the node is simultaneously being driven by a path to power (high or low), then DECSIM ignores the predicted charge share state if you have disabled charge sharing for that node. Enabling or disabling charge sharing does not affect the calculations of charge sharing events when nodes are floating because the charge sharing occurs despite the disabling.

Format

\[ \text{NO\_CHARGE\_SHARE \ signame [\, signame, ...]}; \]

Example

\[ \text{NO\_CHARGE\_SHARE SIG1;} \]
\[ \text{NO\_CHARGE\_SHARE SIG1<7:0>;} \]
\[ \text{!All eight bits of SIG1 are disabled} \]

Placing a NO_CHARGE_SHARE statement on a signal forces that signal to be modeled as a node (never modeled as internal to a stack).

Resolving Multiple Declarations at Different Levels

Once a node has the NO_CHARGE_SHARE attribute, it keeps that attribute at any higher level. NO_CHARGE_SHARE does not need to be respecified at the higher level.

3.6.7.9 TIMEOUT Declaration in MOS Primitives

Every MOS node has a timeout attribute associated with it. Timeout is the amount of time a node stores, or holds, charge before decaying to the logic state U. The TIMEOUT declaration specifies an explicit timeout value for a node. If a timeout is not specified for a node, and DEFAULT_TIMEOUT is not used, the node's timeout defaults to the value specified in the Process Parameter File.

Format

\[ \text{TIMEOUT signame = time [\, signame = time, ...]}; \]
\[ \text{TIMEOUT signame = INFINITY [\, signame = time, ...]}; \]

See Section 3.10.1 for time units. The default time unit is nS (nanoseconds)
Example

```
TIMEOUT SIG1 = 10; !Defaults to 10 NS (nanoseconds)
TIMEOUT SIG1 = 10 NS; !Units specified explicitly
TIMEOUT SIG1 = INFINITY;
TIMEOUT SIG1<7:0> = 10 NS; !All eight bits of SIG1 get the value
```

Note: For specifying delays in MOS models built with equation primitives, see Section 3.6.6.

Placing a TIMEOUT statement on a signal forces that signal to be modeled as a node (i.e. never modeled as internal to a stack).

To avoid simulation scheduling overhead, set TIMEOUT to INFINITY until the network is entirely debugged. Then assign explicit timeouts and test the network to see that it still functions properly. This partitioning ensures that the overhead of timeout scheduling is not carried through the entire design process.

Resolving Multiple Declarations at the Same Level

If the TIMEOUT declaration is given more than once for a signal, the minimum time value is used and a warning is issued. Timeouts set by the TIMEOUT declaration take precedence over the DEFAULT_TIMEOUT network declaration.

Resolving Multiple Declarations at Different Levels

A TIMEOUT declaration for a signal in a higher-level model overrides the timeout value declared on the lower-level model header signals. See Section 3.7.2 for more information.

3.6.7.10 EXPLICIT_NODE Declaration

The EXPLICIT_NODE declaration forces DECSIM to model a signal as a MOS node. This is useful for modeling signals that would otherwise be modeled as internal to a stack (i.e. only nodes are observable from the command language, can be forced to a state, etc.).

Format

```
EXPLICIT_NODE signame [, signame, ...];
```

Example

```
EXPLICIT_NODE SIG1;
EXPLICIT_NODE SIG1<7:0>; !All eight bits of SIG1 get the setting
```
Resolving Multiple Declarations at Different Levels

Once a node has the EXPLICIT_NODE attribute, it keeps that attribute at any higher level. EXPLICIT_NODE does not need to be respecified at the higher level.

3.7 NETWORK-LEVEL DECLARATIONS

The statements described in this section specify network defaults. They are placed in network format (top-level) files, and specify defaults that apply to the entire network, not to just one primitive or node in the network.

3.7.1 PRECISION and RESOLUTION Statements

During simulation, time is divided into discrete time units defined with the RESOLUTION statement. If the resolution is too large, the network's smallest delays are truncated to zero; if the resolution is too small, the simulation may run inefficiently and may cause arithmetic overflow. The resolution can be specified by the RESOLUTION statement or the PRECISION statement. If neither is specified, the default is PRECISION 2.

3.7.1.1 RESOLUTION Statement

The RESOLUTION statement specifies the resolution. A warning is produced at compile time if the specified resolution caused the network's largest delay to overflow, or the smallest delay to be truncated to 0. For example, if the smallest delay for a network is 1 NS and a resolution of 10 NS is given, then the 1 NS delay is truncated to 0 and produces an error.

Format

RESOLUTION time;

Example

RESOLUTION 10 NS;
RESOLUTION 1; !Default is NS.

3.7.1.2 PRECISION Statement

The PRECISION statement specifies the number of digits that the smallest delay in the network retains.

Format

PRECISION integer; !where the integer is from 1 to 7

For example, if the smallest delay is 123456 PS and the precision is specified as 3, then that delay is stored as 123 with a resolution of 10**-9. If a precision of 6 is specified, it is stored as 123456 with a resolution of 10**-12.

The default PRECISION is 2. If the smallest delay is 1NS, that delay is stored as 10 with a resolution of 10**-10 (100PS).
In MOS networks, the smallest RC time-constant determines the smallest delay. In equation networks, the smallest nonzero minimum delay determines the smallest delay. If all delays are zero, RESOLUTION is one nanosecond.

3.7.1.3 Scoping of PRECISION and RESOLUTION
Scoping describes a situation where a lower-level RESOLUTION or PRECISION statement overrides one at a higher level; it is identical for both statements. RESOLUTION and PRECISION statements can appear in a network description at the network level, model level, or subblock level. However, they have meaning only if used with top-level models (models that can be loaded for simulation). There must be only one PRECISION statement or one RESOLUTION statement at the same level. If no PRECISION or RESOLUTION is specified for a given subblock, then the resolution is automatically calculated using a default precision of 2. If both PRECISION and RESOLUTION are specified for a subblock, the RESOLUTION statement takes preference.

Example

!RESOLUTION scoping (PRECISION is identical)
RESOLUTION 1 NS;  ! At network level
MODEL M1 ....;
    GLOBAL;
        RESOLUTION 10 NS;  ! At model level
    ENDDO
    STRUCTURE S1;
        RESOLUTION 100 NS;  ! At subblock level
            ! Resolution for this subblock is 100 NS
        ENDDO
    ENDDO
    STRUCTURE S2;
            ! Resolution for this subblock is 10 NS
    ENDDO
    ENDS
    ENDMODEL M1;
MODEL M2 ....;
    STRUCTURE S1;
            ! Resolution for this subblock is 1 NS
    ENDDO
    ENDS
    ENDMODEL M2;

3.7.2 DEFAULT_TIMEOUT Declaration
The DEFAULT_TIMEOUT declaration supplies a default timeout value for all signals that do not have individual TIMEOUT declarations.
Format

DEFAULT_TIMEOUT time;
DEFAULT_TIMEOUT INFINITY;

See Section 3.10.1 for time units.

Timeouts set by the TIMEOUT declaration take precedence over DEFAULT_TIMEOUT. DEFAULT_TIMEOUT is meaningful only in a network description (not a library model). Any signal without its own TIMEOUT declaration is assigned the DEFAULT_TIMEOUT value. Omitting DEFAULT_TIMEOUT sets all undeclared nodes to the default value specified in the Process Parameter File. If DEFAULT_TIMEOUT is specified at a lower level in a network, it is ignored. Resolving Multiple Declarations at the Same Level

If multiple DEFAULT_TIMEOUT declarations are specified at the network level, the minimum time is used, and a warning is issued.

Note: DEFAULT_TIMEOUT is only used in MOS models.

3.7.3 MAXRES Declaration

The MAXRES declaration supplies the value against which the simulator compares all static conductances during MOS node evaluation. If the computed conductance is greater than the MAXRES value, the network is considered an open circuit for the capacitance calculations. This affects charge sharing and delay calculations. The MAXRES statement is only meaningful at the network level. If you do not specify MAXRES, NETPRO uses the default MAXRES specified in the Process Parameter File.

Format

MAXRES resistance;
MAXRES INFINITY;

See Section 3.10.3 for resistance units. Resolving Multiple Declarations at the Same Level

If NETPRO uses multiple MAXRES statements at the network level, it chooses the minimum resistance value, with warning messages displayed where each new statement occurs.

3.8 Overview of Using Parameters

This section describes how to use NETPRO’s language features to pass parameter values to the various primitive types.
3.8.1 Parameter Types for Primitives

Each primitive has different types of parameter values that can be specified.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Parameter Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equation</td>
<td>RISE–FALL propagation delays (such as RISE0_1)</td>
</tr>
<tr>
<td>Mosfet</td>
<td>TYPE, WIDTH, LENGTH</td>
</tr>
<tr>
<td>Resistor</td>
<td>RESISTANCE</td>
</tr>
<tr>
<td>Memory</td>
<td>Delays (TCER, TCEW) and DISABLE_STATE</td>
</tr>
<tr>
<td>Behavior</td>
<td>Delays in assignment, ACTIVATE, and WAIT statements</td>
</tr>
</tbody>
</table>

You can assign values to these parameters by directly including an explicit value in the primitive definition.

Example

EQUATION [RISE0_1 = 10 NS, FALL0_1 = 20 NS] OUT = NOT IN;

This method is quite restrictive, however, since you must specify the fixed, explicit value at the point where the primitive is defined.

A more flexible method of specifying values for the primitive parameters is described in the following sections.

3.8.2 Formal and Actual Parameters

A formal parameter is a place holder, or variable, for which you can substitute actual values. A formal parameter is defined in the model header. It is normally defined with a default value. Its scope extends over the body of the model, and it can be used anywhere within that model. See Section 3.3.1.1 for more details.

Example

MODEL M1 [P1 = 10, P2 = 20] OUT = IN;

... ! (the 10 NS and 20 NS values are the default values).

... ! Parameters P1 and P2 can be used in statements

... ! here.

ENDMODEL M1;

An actual parameter appears in an instance statement. The value of the actual parameter is passed to the matching formal parameter. An actual parameter can be name–based or order–based, but must match a formal parameter defined in the model definition being instantiated. The value on the actual parameter overrides the default value on the matched formal parameter. Actual parameters must always be supplied when there is no default parameter. See Section 3.4.4 for more details.
Example

\begin{align*}
I1: & \quad M1 [30, 40] \quad \text{OUT} = \text{IN}; \quad \text{! Order-based} \\
I2: & \quad M1 [P1 = 30, P2 = 40] \quad \text{OUT} = \text{IN}; \quad \text{! Name-based}
\end{align*}

Formal parameters and actual parameters are used together to pass parameter values into primitives in models. Thus, when you have multiple instances of the same model, you can specify different parameter values for the primitives defined in those models.

### 3.8.3 A Simple Example of Passing Parameters into a Model

The example below shows how actual and formal parameters are used to pass parameter values into a model. The equation primitive is used here.

**Example**

MODEL M1 [PARAM1 = 10, PARAM2 = 20, 
PARAM3 = 30, PARAM4 = 40] \ O = I1, I2;

\begin{itemize}
  \item The formal parameters (PARAM1, PARAM2, etc.) are used in the equation statement to pass the delay values to the equation primitives.
  \item EQN [RISE0_1 = PARAM1, FALL1_0 = PARAM2] \ O = I1 AND SIG; \quad \text{! First gate}
  \item EQN [RISE0_1 = PARAM1, FALL1_0 = PARAM2] \ SIG = NOT I2; \quad \text{! Second gate}
\end{itemize}
ENDMODEL M1

\begin{itemize}
  \item First instance, actual parameter values are supplied for all 4 formal parameters from M1.
    \item Result: All default parameter values from M1 are overridden.
      \item First gate in M1 has RISE0_1 = 50 NS, FALL1_0 = 60 NS.
      \item Second gate in M1 has RISE0_1 = 70 NS, FALL1_0 = 80 NS.
  \item I1: \quad M1 [PARAM1=50, PARAM3=70, PARAM2=60, PARAM4=80] ... \\
    \item Second instance, no actual parameters.
      \item Result: All 4 formal parameters get set to their default values.
      \item First gate in M1 has RISE0_1 = 10 NS, FALL1_0 = 20 NS.
      \item Second gate in M1 has RISE0_1 = 30 NS, FALL1_0 = 40 NS.
  \item I2: \quad M1 ... \\
    \item Third instance, 2 actual parameter values supplied.
      \item Result: The other 2 formal parameters get set to their default values.
      \item First gate in M1 has RISE0_1 = 50 NS, FALL1_0 = 20 NS.
      \item Second gate in M1 has RISE0_1 = 70 NS, FALL1_0 = 40 NS.
  \item I3: \quad M1 [PARAM1=50, PARAM3=70] ... ; \quad \text{! PARAM2 and PARAM4 are defaulted}
\end{itemize}

### 3.8.4 Primitive Parameters That Can Be Passed

Not all of the various types of parameters for the primitives can have values passed into the model where they are defined. The following table describes which parameters can be passed. The sections after the table describe restrictions.
### Table 3-6  Primitive Parameters That Can Be Passed

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Parameter Type</th>
<th>Able to Pass Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equation</td>
<td>RISE–FALL delays</td>
<td>Yes</td>
</tr>
<tr>
<td>Mosfet</td>
<td>TYPE</td>
<td>No – TYPE has a text value; only numerics can be passed.</td>
</tr>
<tr>
<td></td>
<td>WIDTH</td>
<td>Yes, with restrictions.</td>
</tr>
<tr>
<td></td>
<td>LENGTH</td>
<td>Yes, with restrictions.</td>
</tr>
<tr>
<td>Resistor</td>
<td>RESISTANCE</td>
<td>Yes, with restrictions.</td>
</tr>
<tr>
<td>Memory</td>
<td>Delays (TCER, TCEW)</td>
<td>Yes</td>
</tr>
<tr>
<td>Behavior</td>
<td>DISABLE_STATE</td>
<td>No – This is a logic state.</td>
</tr>
<tr>
<td></td>
<td>Delays in assignment, WAIT, and ACTIVATE statements.</td>
<td>Yes</td>
</tr>
</tbody>
</table>

#### 3.8.5  Passing Parameters Through Multiple Levels

To pass parameter values down more than one level, you can use placeholders, called formal parameters, in instance statements and equation statements. There is no limit to the number of levels through which a parameter value can be passed.

The following example illustrates passing values through two levels.

**Example**

```
!This model defines two formal parameters that are used inside
!the model as parameters for an equation statement.
MODEL AND [RISE_PARAM = 2, FALL_PARAM = 3]  O = I;
EQN [RISE0_O = RISE_PARAM, FALL1_O = FALL_PARAM] O = NOT I;
ENDMODEL NOT;

MODEL halfadd [par1 = 20ps, par2 = 20ps] sum = a, b;
    A1: and [par1, par2] sum = a, b;  !Rise and fall delays of
    ENDMODEL halfadd;  !20ps.
MODEL adder [par3 = 50ps] sum1, sum2, sum3 = a1, b1, a2, b2, a3, b3;
I1: halfadd [par1 = par3, par2 = par3] sum1 = a1, b1;  !Delays are overridden;
    I2: halfadd [par1 = 30ps] sum2 = a2, b2;  !Only the rise delay is
    !changed; the fall delay !remains 20ps.
I3: halfadd sum3 = a3, b3;  !Default delays defined in model halfadd
    !are used: rise and fall of 20ps.
ENDMODEL adder;
```

#### 3.8.6  Handling Min, Nom, and Max Values (Parameter Triples)

Not only can simple parameter values (like “10 NS”) be specified, but DECSIM supports the specification of a full triple–value quantity.
Interconnect Language

Format

minimal-value : nominal-value : maximum-value

The maximum-value must be greater than or equal to the nominal-value. The nominal-value must be greater than or equal to the minimum-value. The minimum-value must be greater than or equal to zero.

These triple-value quantities can be used anywhere where a parameter value can be used:

- In primitive definitions, for example: EQUATION [RISE0_1 = 10:20:30, FALL1_0 = 40:50:60] ... = ... ;
- As formal parameter default values: MODEL M1 [PARAM_1 = 10:20:30, PARAM_2 = 40:50:60] ... = ... ; . . ENDMODEL M1;
- As instance actual parameter values: I1: M1 [PARAM_1 = 10:20:30, PARAM_2 = 40:50:60] ... = ... ;

3.8.6.1 DECSIM Selecting of Min, Nom, Max Values (SET TIMING Command)
When your network is compiled, NETPRO computes all three values (min, nom, max) for each primitive parameter and stores this data in the .NOB file. When the network is loaded into DECSIM for simulation, you can select which parameter set you want loaded by using the SET TIMING command. The command effects the loading of the network the next time you use the COMPILE or USE command. Some points:

- With the SET TIMING command, you can only select MIN, NOM, or MAX for the entire network. There is no way to specify different selections for different primitives in your network.
- If you do not use SET TIMING, DECSIM selects MAXIMUM.

3.8.6.2 Interpolating Missing Values
It is valid to leave parts of a triple unspecified:

Example

```
    100    ! A single value is really a very small triple
   10:20  ! Nom value is unspecified
   30:40  ! Max value is unspecified
```

In these cases, NETPRO automatically interpolates the missing min, nom, or max values.

The following table specifies how this interpolation is implemented. In the table, the letter V represents a value. The missing values are referred to by their position: min, nom, or max.
Table 3-7  Interpolating Min, Nom, and Max Values

<table>
<thead>
<tr>
<th>Form Specified</th>
<th>Equivalent Form</th>
<th>Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>V:V:V</td>
<td>None needed (all values present)</td>
</tr>
<tr>
<td>V1:V2</td>
<td>V1::V2</td>
<td>Nom = (V1min + V2max)/2</td>
</tr>
<tr>
<td>V1:</td>
<td>V1:V1min:V1min</td>
<td>Nom = max = V1min</td>
</tr>
<tr>
<td>:V2</td>
<td>0::V2</td>
<td>Nom = V2max/2, min = 0</td>
</tr>
<tr>
<td>:</td>
<td>0</td>
<td>ERROR</td>
</tr>
<tr>
<td>::</td>
<td></td>
<td>ERROR</td>
</tr>
<tr>
<td>V1:V2:V3</td>
<td>V1:V2:V3</td>
<td>None needed (all values present)</td>
</tr>
<tr>
<td>V1:V2:</td>
<td></td>
<td>Max = (2 * V2nom) - V1min</td>
</tr>
<tr>
<td>V1::V2</td>
<td></td>
<td>Nom = (V1min + V2max)/2</td>
</tr>
<tr>
<td>V1::</td>
<td></td>
<td>Nom = max = V1min</td>
</tr>
<tr>
<td>:V1:V2</td>
<td></td>
<td>Min = (2 * V1nom) - V2max</td>
</tr>
<tr>
<td>:V1:</td>
<td>0:V1:</td>
<td>Max = (2 * V1nom), min = 0</td>
</tr>
<tr>
<td>::V1</td>
<td>0::V1</td>
<td>Nom = V1max/2, min = 0</td>
</tr>
</tbody>
</table>

3.9  Compiler Lexical Statements

3.9.1 REQUIRE

The REQUIRE statement incorporates a specified file into your network description.

Format

REQUIRE  filename.ext  [ , ... ];

The default extension file type is .NRQ. The file name conforms to the host operating system conventions and must be enclosed with single quotation marks if it contains a semicolon.

The contents of the specified source file are incorporated into the network description at the point of the REQUIRE statement. Such a file usually contains a network description, a set of model definitions, or both.
REQUIRE facilitates the development of modular networks, the use of standardized networks such as subprocessors and terminals, and the inclusion of commonly used network models in more than one network description.

A REQUIRE file can contain another REQUIRE statement. The maximum nesting possible varies according to the host system open-file limit and the number of library files referenced, but it is always at least three.
3.9.2 **INFORM**

The INFORM statement checkpoints the progress of your network description file during compilation. INFORM statements can be placed anywhere in your network. The text message is printed as soon as it is encountered.

**Format**

```
INFORM 'text to be printed on tty'
```

As NETPRO compiles a network and encounters the inform command, it sends a message to the terminal. The inform command terminates with a semicolon.

**Example**

```
INFORM 'finished processing elements of chip 1';
```

3.9.3 **Text Macros**

You can define text string macros anywhere in the network description file, as long as they are defined before they are used. These macros are purely lexical and are substituted (expanded) before any network description language grammar semantics are inferred. Text macros can have parameters (that is, be called and defined within square brackets), and all text macros must end with the keyword $ENDMACRO. The MACRO definition extends to the end of the current model definition level. The MACRO scope includes lower-level model definitions. MACROs can be nested, but all MACRO – $ENDMACRO pairs must match. All text in the definition is taken as the literal text for substitution. There can be no space after the $ in $ENDMACRO.

**Note:** Do not use the keywords MACRO, ENDMACRO, or $ENDMACRO inside the macro body, not even within the text of a %PRINT statement.

**Format**

```
-- for definition:
MACRO name [formal list] = macro body $ENDMACRO;
MACRO name [ formal = 'default', ... ] = macro body $ENDMACRO;
MACRO name = macro body $ENDMACRO;
-- for use:
macroname
macroname [ 'actual' [ , ... ] ]
macroname [ formal = 'actual' [ , ... ] ]
```

**Note:** The use of single quotation marks surrounding formal default and actual values is a temporary implementation restriction that will be removed at a later date.
3.9.3.1 String Manipulation

If the default is omitted from the definition, the default becomes the null string. When the MACRO is used, name-based parameters that are omitted will be assigned the default value, and order-based parameters will be assigned the default value if the parameter position is null.

The macro body can be multiline, and the formal list is optional. Everything, including comments, text strings, and the keywords MACRO and $ENDMACRO, is parsed for formal parameter substitution. When you are defining a macro, an equal sign must be used to separate the macro name from the contents of the definition. The $ is used as a concatenation operator in the definition for the text and then disappears. To redefine a macro, quote its name in the new definition.

Example

MACRO a = Defn #1 $ENDMACRO;
MACRO 'A' = Re-definition $ENDMACRO;  !first macro definition is
 !thrown away and superseded

When you use the macro, macro names cannot be imbedded within a number or signal name for the purpose of macro expansion. A quoted name, either single or double quotation marks, is never taken as a macro expansion request.

3.9.3.2 Text Concatenation

Text concatenation is needed in macro definitions and calls to allow formation of signal names and other lexemes from bits and pieces of strings (literal and formals). The "$" operator performs this splicing of text strings and provides a break character to give the parser a chance to substitute the formal name. All unquoted dollar signs vanish after splicing.

Example

MACRO SLICE [N] = C$N=P$N,G$N $ENDMACRO  !Definition
7400 SLICE[0];  ! expands to:  7400 C0=P0,G0;
7400 SLICE[1];  7400 C1=P1,G1;
...
7400 SLICE[7];  7400 C7=P7,G7;

3.9.4 REVISION Statement

The REVISION statement specifies a revision number of a model definition. Include a REVISION statement in every model definition to update the revision number each time the definition is modified. A comment describing the modification is also helpful, as the accumulation of comments provides a revision history of the model.
3.10 Units

This section summarizes all the units that can appear in the network description language and the Process Parameter File.

Each set of units specifies a value (a real number) in a particular context. For example: a resistance value might be 1 K, a time might be 9.5 NS, a distance might be 12 (where the units default to UM).

3.10.1 Time Units

The explicit time units are:

- M = 1E6 seconds
- K = 1E3 seconds
- S = 1 second
- MS (milliseconds) = 1E–3 seconds
- US (microseconds) = 1E–6 seconds
- NS (nanoseconds) = 1E–9 seconds
- PS (picoseconds) = 1E–12 seconds
- FS (femtoseconds) = 1E–15 seconds

Default time units are NS (nanoseconds).

3.10.2 Capacitance Units

The explicit capacitance units are:

- UF (microfarads) = 1E–6 farads
- NF (nanofarads) = 1E–9 farads
- PF (picofarads) = 1E–12 farads
- FF (femtofarads) = 1E–15 farads

Default capacitance units are PF (picofarads).
3.10.3 Resistance Units

The explicit resistance units are:

- M (megohm) = 1E6 ohms
- K (kilohm) = 1E3 ohms
- Ω = 1 ohm

Default resistance units are Ω (ohms).

3.10.4 Distance Units

The explicit distance units are:

- MM (millimeters) = 1E−3 meters
- UM (microns) = 1E−6 meters
- NM (nanometers) = 1E−9 meters

Default distance units are UM (microns).

3.10.5 Area Units

The explicit area units are:

- SQ-MM (square millimeters) = 1E−6 square meters
- SQ-UM (square microns) = 1E−12 square meters
- SQ-NM (square nanometers) = 1E−18 square meters
Part 2  Compiling and Simulating Models
This chapter presents the general format of DECSIM commands and the rules for entering the commands. Detailed information on all the DECSIM commands is presented in Chapter 5.

DECSIM Initialization

When DECSIM is invoked, DECSIM$SYS:DECSIM.SYS is executed. This file is provided when DECSIM is installed. It contains informational messages and system macro definitions.

You can also have a personal initialization file (DECSIM.INI) which contains functions that you want executed every time you run DECSIM, such as opening log files or defining macros. DECSIM looks for this file in your current directory. If you want one DECSIM.INI file that can be accessed from any directory, define DECSIM$INIT in your login.com file to point to the DECSIM.INI file. For example:

```
$ DEFINE DECSIM$INIT doc$disk:[user.project]decvsim.ini
```

With the /NOINITIALIZE DCL command line qualifier, you can specify that the DECSIM$INIT and DECSIM.INI files not be executed. With the /INITIALIZE = (file,...) qualifier, you can specify another file or set of files to be executed. Assuming that DECSIM is a symbol defined as "$DECSIM$SYS:DECSIM.EXE"

1 Type DECSIM and DECSIM$SYS:DECSIM.SYS, the file pointed to by the logical DECSIM$INIT, and DECSIM.INI are all executed.

2 Type DECSIM/NOINI and only DECSIM$SYS:DECSIM.SYS is executed.

3 Type DECSIM/INI = (file,...) and DECSIM$SYS:DECSIM.SYS plus any files specified in the INI list are executed.

Three other qualifiers can be used with the DECSIM command:

- /CPUTIME = delta_time limits the amount of CPU time for a DECSIM run.
- /INPUT = input_file redirects SYS$INPUT to the specified source. Commands in the input_file will be executed immediately.
- /OUTPUT = output_file redirects SYS$OUTPUT to the specified destination. Output goes to the specified file, rather than to the terminal.
4.1 Command Language Conventions

The following sections describe command language syntax and entry.

4.1.1 Command Format

A DECSIM command consists of either a single command or a BEGIN-END block of commands. The general format is as follows:

**Format**

[prefix] verb [/global_qualifier] [argument] [/local_qualifier] ;

**prefix**

A control prefix is an optional control or iterative prefix expression, such as IF-THEN, WHILE, or FOR.

A variant prefix is an optional variant prefix, selected from SET, CANCEL, SHOW, ENABLE, or DISABLE.

**verb**

A command verb, such as DEPOSIT or WATCH.

**/global_qualifier**

An optional qualifier to the command verb. The qualifier to the command verb applies to every argument or symbol in the command. All qualifiers, both global and local, use an equal sign to join them to arguments. You can also use a colon.

**Example**

    /qualifier= argument
    /qualifier: argument

**argument**

An access list (list of symbol names separated by commas), a number, or a keyword.

**/local_qualifier**

An optional qualifier to a symbol or device. The local qualifier affects only the device or symbol that it modifies. All qualifiers use an equal sign to join them to arguments. You can also use a colon.

**Example**

    /qualifier= argument
    /qualifier: argument

; (semicolon)

An optional command terminator. Use either a semicolon or <CR> (carriage return) to terminate commands. Use semicolons to separate multiple commands entered on one line.
Sometimes the command may be too long to fit on one line, or you may want to enter a comment. In this case, the syntax could look like the following:

**Example**

```
[control_prefix] [variant_prefix] verb [/global_qualifier] -
!
the dash in the above line is for command line continuation
[/global_qualifier] [access_list] /local_qualifier] ;
```

- **(dash)**

A line continuation (dash) is needed to continue a command from one line to the next if the command could end at the carriage return. For example, when an optional qualifier is on a separate line from what it modifies, you must use line continuation. However, a dash continues the command for one line only. In this example, the command is not completed until the third line. To ensure line continuation, the comment (which DECSIM ignores) must be preceded by a dash.

**Note:** A line continuation (dash) is NOT needed if the command could not end at the carriage return. For example, if the command required a file name as an argument, DECSIM would not accept the command without the required file name.

See Section 4.1.4 for more information.

- **(exclamation mark)**

A comment is indicated by a preceding exclamation mark. DECSIM ignores everything from the exclamation mark to the end of the line. See Section 4.1.3 for more details.

## 4.1.2 Abbreviations

The DECSIM command language allows abbreviations for certain keywords and symbols. However, in the preparation of indirect command files, the use of full, unabbreviated keywords and symbols helps to avoid later conflicts and ambiguities resulting from signal name changes and new macro definitions.

### 4.1.2.1 Command Keyword Abbreviations

Command names, macro names, qualifier names, and symbol names can be abbreviated to the minimum number of characters that distinguish the name.

For example, the TYPE command can be abbreviated TY. However, the EXIT command requires three characters (EXI) to distinguish it from the EXAMINE command.

The most frequently used DECSIM keywords can be specified by typing one letter. Those keywords are:

```
E EXAMINE
H HELP
S SIMULATE
D DEPOSIT
Q QUIT
T TIME
```

You can enter these keywords by typing a one-letter abbreviation, by typing the minimum number of characters that will distinguish one keyword from other keywords, or by typing the full keyword.
The following command keywords cannot be abbreviated:

BEGIN END
IF THEN ELSE
WHILE UNTIL DO
FOR FROM TO BY
LEAVE
SELECT FROM ALWAYS OTHERWISE
FS FS NS US MS S (time units)
expression operators (AND, OR, etc.)

Note: In a macro definition, you can override the normal abbreviation procedures and specify the minimum number of characters needed to call that particular macro.

4.1.2.2 Symbol Name Abbreviations
While using DECSIM interactively, you can abbreviate the symbol names used in conjunction with commands such as TRACE, EXAMINE, and DEPOSIT.

4.1.3 End–of–Line Comment (!)
Comments can be attached to the end of any command line except line–continued strings by preceding the comment with an exclamation mark.

Format

! comment text

Description

An end–of–line comment is identified by an exclamation mark (!). Except for listings and echoing, everything that appears between the "!" and the next line–feed character is a comment and is ignored by DECSIM. Any keyboard character other than carriage return, line feed, or vertical control characters can be used in a comment.

Comments are not recognized within a text string enclosed in quotation marks (see Section 4.2.1), but are recognized after the dash that indicates line continuation. See Section 4.1.4 for more information. End–of–line comments cannot be continued; an exclamation mark is needed for each comment line.
4.1.4 End-of-Line Continuations

A statement or command sequence too long to fit on one line can be continued by ending all lines except the last with a dash (minus sign, "-").

Note: This differs from the interconnect and behavior languages where only a semicolon can end a statement.

A minus sign is simply a minus sign, unless it appears as the last printing character on a line.

To use line continuation for a line that already ends with a dash, type another dash.

You can follow a line continuation character with spaces, tabs, or comments. You can also use leading spaces and tabs in continued lines for clarity.

To end a command with a minus sign, place a semicolon after the minus sign.

Line continuation characters are not recognized in strings or in comments.

Multiline strings and comments require special syntax. Strings are continued to the second quotation mark, regardless of dashes and carriage returns within the quotation marks. Comments are preceded on each line by an exclamation mark.

Where the command could not possibly end, a line can be continued without the use of a line continuation character. For example, you don’t need a dash for a line ending with DO or EXAMINE because both are incomplete commands.

See Section 4.1.1 for more information.

4.1.5 Labels (Command Labels)

This section describes labels as they are used for a block or statement. It does not describe instance labels. See Section 3.4.1 for information on instance labels.

Format

```
label: !label definition (for example, label: BEGIN ... END)
label !label usage - argument of command (for example, LEAVE label)
```

Note: The colon must follow the label name definition with no intervening spaces or carriage returns.
A label tags a block or control flow prefix for exiting that block. Labels are defined before the statement and must be followed directly by a colon. A label is used in a LEAVE statement for premature block or statement exit.

The label can be any name beginning with an alphabetic character and containing letters, digits, and underscore (_). However, the label name "ALL" is reserved and cannot be redefined—it has special meaning for the LEAVE command.

Labels are global, and two labels of the same name should not be used at the same time. However, once a block or statement is exited (or a WATCH is canceled), the label name can be reused.

See the LEAVE command in Chapter 5 for more information on labels.

---

### 4.1.6 Errors

There are four types of error message:

- `%I-xxx, message !informational`
- `%W-xxx, message !warning`
- `%E-xxx, message !error`
- `%F-xxx, message !fatal`

#### Description

The four types of DECSIM error messages provide you with debugging hints when you type an invalid command or try to simulate a model with faulty logic.

Informational messages act as reminders of how DECSIM interprets a command and of any possible side effects.

Warning messages indicate that what you have done is questionable and that the command, by ignoring data or substituting default data, will be executed anyway. For example, warning messages can indicate that you may have mistyped a command, or that a declaration or default is not what is expected. Warning messages do not necessarily mean that anything is wrong.

Error messages indicate that DECSIM did not accept the command. Possible causes include incorrect syntax and invalid names.

Fatal errors terminate DECSIM execution.

---

#### 4.1.6.1 Error Handling of Syntax Errors

Syntax errors occur when you misspell a keyword or use an incorrect language construct. The error message reports the problem and the attempted solution.

**Example**

```plaintext
SIM> Examine SIG1, SIG2 GARBAGE
%W, [SIMSER] Syntax error.
Examine SIG1, SIG2 GARBAGE

Problem: "GARBAGE" is a NAME that is not valid here.
Attempted correction: "GARBAGE" ignored.
```
With the ERROR command, you control how DECSIM displays error messages and what DECSIM does when an error occurs (for example, abort or continue). See Chapter 5 for a description of the ERROR command. How DECSIM handles syntax errors depends on the problem:

- If you type the syntax error at the terminal, DECSIM ignores everything after the error (as indicated by the error diagnostic message).

- If the syntax error occurs during the execution of an indirect file or the expansion of a macro definition, the error message includes information concerning the file or macro name, line number, and incorrect line contents. The rest of the offending line will be ignored, including continuation lines that are indicated by a dash, and a PAUSE will be executed to return the user to interactive mode. If the job was executing in batch mode, the batch job may be terminated depending on the setting of the ERROR command.

- If the syntax error occurs at the end of the expansion of a macro or reading of an indirect file, DECSIM may display the MORE> prompt. This shows that the syntax error during expansion was not resolved, but that DECSIM will allow you to complete the command.

After a syntax error message, you can use control G (CTRL/G), then retype the entire command. CTRL/G brings you back to top level command mode and aborts all levels of looping.

4.1.6.2 Semantic Error Handling
Usually, the error messages for semantic errors indicate what is assumed or what action is taken to recover. For instance, if you divide by zero, DECSIM assumes a zero result for the operation. If you specify an invalid signal name in a list for an EXAMINE or TRACE command, DECSIM ignores it, but keeps the rest of the list intact.

4.1.6.3 Error Handling for Expressions, EVALUATE
There are two consequences of errors in expressions and in EVALUATE assignment statements:

- If, in an expression, you use a variable name that is undefined or invalid, the expression operand value is assumed to be 0. (This is reported in the error message.)

- If you try to assign a value to an undefined or invalid variable with the EVALUATE command, a new command language STATE is implicitly defined and the EVALUATE expression value assigned to it. This action is reported in the error message. As a result, and because command language states override behavior states, if you try to assign a value to an invalid behavior state, DECSIM implicitly defines a command language state and assigns the value to it.
4.1.6.4 Error Reporting During Indirect File Execution

If a syntax or semantic error occurs during the execution of an indirect file, the error is reported in the following way:

```plaintext
%W, [SIMSER] Syntax error.
    In file xxxxxxx.yyy;1 line 2, page 1.
Examine SIG1, SIG2 GARBAGE

Problem: "GARBAGE" is a NAME that is not valid here.
Attempted correction: "GARBAGE" ignored.
```

4.1.6.5 Error Reporting During Execution of Blocks and Loops

The input source command line indicated by the error message for an error in a BEGIN-END block or control flow block is always the last line of the block, even though the error may have occurred elsewhere in that block. This happens because all the commands within a block or loop structure are collected (and parsed) before any of them are executed. The following examples illustrate two versions of the same error: Example 1 occurs in an indirect file; Example 2 occurs interactively.

**Example 1**

```plaintext
SIM> @ERROR/ECHO
SIM> BEGIN
MORE> EVALUATE none !<<<< ERROR
MORE> PRINT 'PRINT line executed'
MORE> END
%E, [SIMION] Expr operand not defined; assumed 0 value for operand
named:
NONE>>>>END
While reading from
1 INDIRECT DISK$:GENERAL:[TEMP.PHILLIPS]ERROR.IND;1 Line 4/1
0
PRINT line executed
! End of indirect file DISK$:GENERAL:[TEMP.PHILLIPS]ERROR.IND;1
```

**Example 2**

```plaintext
SIM>
SIM> BEGIN
MORE> EVALUATE none
MORE> PRINT 'PRINT line executed'
MORE> END
%E, [SIMION] Expr operand not defined; assumed 0 value for operand
named:
NONE 0
PRINT line executed
SIM>
```

4.2 Command Arguments

The following sections describe items, such as files, logic values, and expressions, that are used as command arguments.
DECSIM Environment and Conventions

4.2.1 Quoted Text Strings

DECSIM supports two types of quoted text:

- 'quoted text strings' with single quotation marks, and
- "quoted symbol names" with double quotation marks. See Section 4.3.7.2 for more information.

Format

'any text'
"signal or logical names with special characters"

Description

Double quotation marks enclose symbol names that have wildcard characters or special characters.

Text strings within single quotation marks are arguments for certain commands, such as PRINT. Carriage returns are ignored within this text. Control characters are represented either by themselves or by a caret (^) followed by the corresponding alphabetic character. Either upper- or lowercase is accepted. The single quotation mark is represented by a pair of single quotation marks. A caret is represented by a pair of carets.

Single quotation marks can also be used in the following cases:

- By enclosing file names in single quotation marks, you can use special characters such as a semicolon (;).
- By enclosing a macro name in single quotation marks, you can inhibit the macro expansion.

Example

The string:

'"\"I\"Abc\"\"'"

represents:

\"TAB\"Abc\"<CR>

Strings enclosed in single quotation marks can appear in the following commands:

- EXAMINE/SEPARATOR
- LOAD/ID
- LOG/TITLE
- MACRO
- PRINT
- PATTERN/DELIMITER
- TRACE/SEPARATOR
4.2.2 File Specifications and DECSIM Log Names

File name syntax conforms to the standards of the operating system on which DECSIM is being run. The general form is:

```
node:device:[directory.subdir. ...]filename.ext;rev  1VAX/VMS
```

File names can be written in uppercase or lowercase letters and can include numbers. Names can also include the underscore character (_), the hyphen (–), and the percent sign (%), provided that it is not the first character.

The defaults for omitted fields depend both on DECSIM context and the operating system. Wildcards are not allowed in DECSIM file specifications.

File names can appear in the following commands:

```
COMPILE
CONFIGURE
LOAD
LOG
PATTERN
```

4.2.2.1 Special Characters in File Names

If special characters (such as "","") are included in a file name, the entire file name must be enclosed in single quotation marks.

Example

```
SIM> SET LOG/CMD  'OUT.LOG;3'
```

Note: If the VMS file revision number is preceded by a dot (.), no special quoting is necessary. Example: OUT:LOG.3

4.2.2.2 Default Directory and Paths

The default path or directory is chosen for a file lookup when the directory is omitted from the file specification.

4.2.2.3 Default File Extensions

Default file extensions (types) are shown with their respective files throughout this manual.

4.2.2.4 DECSIM Log Names

DECSIM log names are short names for files that the simulator has opened for output.

DECSIM log names are not to be confused with VMS logical name translations for file names. DECSIM log names allow you to attach a "handle" to a log file so that you can easily refer to it from the DECSIM command language without having to specify the full file name. They are applicable only within a DECSIM session. VMS logical names allow you to set up translatable file names so you can assign a specific file name or device to a logical name.

Log names can be specified with the SET LOG command, or with the /LOGNAME qualifier on other commands such as TRACE and PRINT.
DECSIM Environment and Conventions

If you use the /LOGNAME qualifier, and the log name has already been set up with a SET LOG command, output is routed to the file opened by the SET LOG command. If you use the /LOGNAME qualifier, and the log name has not been set up with a SET LOG command, an implicit SET LOG command is executed which opens a file whose file name is the log name and whose extension depends on the command that is writing the file.

Example

SIM> SHOW LOG
%I-NLO, No LOG files open
SIM> TRACE/LOGNAME:T1 O
SIM> SHOW LOG 'The TRACE command has implicitly opened this LOG file
LOG/NOCOMMANDS/NORESPONSES/TRACE/NOECHO/NOCOMMENT/HEADER/NOARROW
/LOGNAME:T1
FILE=DSU$:[HEILMAN]T1.TRA;1

After a log file is opened, any later reference to it must be made by using the log name, not the file name.

Examples

SIM> SET LOG /LOGNAME:A XYZFILE.LOG
'XYZFILE.LOG' is the file name, 'A' is the log name.
SIM> CANCEL LOG A
The log name 'A', not the file name, is used here.

Log names can appear in the following commands:

DEBUG/LOGNAME  PRINT/LOGNAME
DETECT/LOGNAME  SHOW FAULT/LOGNAME
ERROR/LOGNAME   LOG/LOGNAME

4.2.3 Logic Values

DECSIM logic is binary. However, in addition to 1 (H or high) and 0 (L or low), two other signal states are recognized: U (undefined) and Z (high impedance). Logic values can be followed by a radix, which is specified by a number following a pound sign (#). For example, U#2 represents an undefined value in base (radix) 2.

To type a logic value containing U (undefined) or Z (high-impedance) states, you must specify the logic value in base 2 (for example, U#2 or Z#2). Otherwise, DECSIM interprets the logic value as a name of an operand.

Logic values can appear in many places, including:

DEPOSIT
FAULT/STUCK_AT
LOAD files
LOAD/INITIAL
Pattern files
Operands in expressions in a command
The radix of the logic value typed as part of a command is not governed by SET RADIX; it is governed by the radix indicated on the value following the pound sign. The default radix is 10.

In the following two examples, Example 1 includes the EVALUATE expression.

**Example 1**

```
EVALUATE 10 ! interpreted as 10 decimal
EVALUATE 10#2 ! interpreted as 2 decimal
EVALUATE 10#16 ! interpreted as 16 decimal
SET RADIX 2
EVALUATE 10 ! interpreted as 10 decimal; SET RADIX has no effect
```

**Example 2**

```
EVAL U210#2 !This is OK
EVAL U210 !This causes the error message: "Expr operand not defined"
```

Logic values are displayed by many commands in DECSIM (EXAMINE, EVALUATE, PRINT, TRACE, etc.).

How DECSIM displays logic values is governed by the SET RADIX command. The default display radix is 10. Most commands that display logic values allow a /RADIX qualifier on the command, which affects only that command.

**Example**

```
(The display of a STATE called TEST has been initialized to hold the value 10 decimal):
PRINT TEST ! Displays "10" (default radix)
PRINT/RADIX=2 TEST ! Displays "1010" (due to /RADIX)
SET RADIX 16
PRINT TEST ! Displays "A" (due to SET RADIX)
PRINT/RADIX=2 TEST ! Displays "1010" (due to /RADIX)
```

**Using the Wildcard Digit (Question Mark) in Logic Values**

The question mark character (?) can be used as a wildcard digit in some logic values, depending on where the values appear. It can appear only in logic values with radix 2, 4, 8, and 16.

The wildcard digit has two main uses:

- In a PATTERN file or a DEPOSIT waveform, the question mark means "allow this signal to be undriven (FREED)."

  **Example**

  ```
  This command forces the signal to 0 for 100 NS, then frees the signal for 200 NS in a repeating waveform.
  DEPOSIT/FORCE SIG = [0:100, ?#2:200] INFINITY
  ```

- In the SELECT statement, a question mark in a trigger expression means "do a wildcard 'match' for that digit."

4–12
Example

The trigger expression in this example is triggered when the variable \( A<3:0> \) is set to 2, 3, 6, or 7:

```
SELECT A<3:0> FROM
[ 0?1?2 ]; PRINT 'Match';
ENDSELECT
```

If the wildcard digit appears in any other logic values, it is interpreted as undefined (U#2).

### 4.2.4 Integer Numeric Values

Integer numeric values appear in many commands, including:

- `COMPILE/ERROR_LIMIT`
- `DROP`
- `DEPOSIT` (repeat count for waveform)
- `EXAMINE/RADIX`
- `FAULT/PERCENT`
- `PATTERN/WIDTH/RADIX`
- `PRINT/WIDTH/RADIX`
- `RADIUS`
- `TRACE/WIDTH/RADIX`
- `SAVE/KEEP`
- `FROM` and `TO` arguments in a `FOR` loop

The radix of the numeric value being typed as part of a command is not governed by `SET RADIX`; it is governed by the radix indicated on the value with the pound sign. The default type-in radix is 10.

### 4.2.5 Time Values

Both absolute and relative time values can be specified. An absolute time value assumes simulation time 0 as a reference point. A relative time value has the current simulator time as a reference point. Absolute time values must be specified by a preceding vertical bar ( | ). If a time value has no vertical bar prefix, it is relative.

**General Format**

- `time [units]` !relative time
- `|time [units]` !absolute time
- `(time_expression)` !for example, `(3*X)`

where units are selected from: FS FS NS US MS S

Spaces between the time and the time units are optional.

Expressions can be used where time values are legal. The expression must be in parentheses (for example, `SIM>` `SIMULATE (X+Y)`).

Relative time specifications included in macro definitions and watch commands reference the time when the macro is expanded or the watch action executed, not the time when the definition is entered.

Time values can appear in many commands, including:

- `ACTIVATE` SET TIME
- `DEPOSIT` SIMULATE
- `DETECT` TRACE
- `PATTERN` SAVE/AT/EVERY
- `WATCH`

Time arguments are also used in the macro facility's $time template parameter. See the MACRO command in Chapter 5 for more information.
%TIME variables can be used in expressions. Time values are not affected by the set radix command. They are always decimal radix and are represented by unsigned integers.

### 4.2.6 Expression Components

A DECSIM command language expression is a set of operators and operands that evaluates to a value. For a complete description of the operators allowed in the command language, see Appendix D.

**Format**

```
operand
operator operand
operand operator operand
```

where operand can be a constant, variable, or subexpression.

Expressions and subexpressions can have parentheses.

**Example**

```
A
A + 1
1010#2 / B
(3 * (B + 1/C))
```

Expressions appear in many commands, including:

- CALL
- EVALUATE
- IF-THEN-ELSE
- SELECT
- WATCH
- DEPOSIT
- FOR-FROM-TO-BY
- PRINT
- UNTIL
- WHILE

### 4.2.6.1 Expression Operands

Operands can consist of constants, special system variables, symbol names (access list names, abbreviations, but no wildcards) and variables (command states).

### 4.2.6.2 Expression Constants

Constants are literal numbers, not variables, used in expressions and qualifier arguments. A number can stand alone or be a part of an expression as an operand. There are two types of constants:

- Logic values – Numeric bit strings of one or more of the logic values 0,1,Z,U,H,L, and ?, or the numbers 0–9 and A–F in the appropriate radix.
- Numeric integer values – Arguments used for “counting” purposes only. They can contain the numbers 0–9 and A–F in the appropriate radix.

Constants have two characteristics: value and width. The value is made up of one or more bits organized in a bit vector. Each bit can have a value of 0, 1, U, or Z. In certain cases, bit values can be defined with the wildcard. The width of a constant, which affects concatenation and shifting, is the number of bits specified. See Section 3.4.6 for the appropriate rules.
Note: Numbers that look like variable names must have a radix sign (#) following them. For example, EF#16 and ZU#2 are numbers; without the radix sign, EF and ZU are variable names.

Example

SIM> EVALUATE width X

Width is an operator that returns the bit–width of X.

where X is:

<table>
<thead>
<tr>
<th>X</th>
<th>Result (shown in decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1* (depends on current input RADIX setting)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>0#2</td>
<td>1</td>
</tr>
<tr>
<td>0#8</td>
<td>3</td>
</tr>
<tr>
<td>0#10</td>
<td>1</td>
</tr>
<tr>
<td>0#16</td>
<td>4</td>
</tr>
<tr>
<td>5#8</td>
<td>3</td>
</tr>
<tr>
<td>5#16</td>
<td>4</td>
</tr>
<tr>
<td>7FFF#16</td>
<td>16</td>
</tr>
<tr>
<td>00007FFF#16</td>
<td>32</td>
</tr>
</tbody>
</table>

SIM> RADIX 16
SIM> STATE REG <16:0>
SIM> EVALUATE REG = 4
SIM> EVALUATE REG & 0#16
SIM> EVALUATE & 0#16
SIM> EVALUATE width 1#16 & 0#8
SIM> 7

Width and values for expression constants have the following characteristics:

• 256–bit maximum width
• Radix 2 through 16
• Four states and wildcard (question mark) digits

4.2.6.3 System Variables

There are many system variables that can be used as expression operands. These variables cannot be abbreviated. With the PRINT command, the expression must be in parentheses. For example, PRINT %TIME or PRINT (%TIME + 1).

%ARRAY_ELEMENT

The element number of the last element changed for an array. If the last element changed is A[2], %ARRAY_ELEMENT contains 2.

%BADCANCEL

Fault events scheduled but canceled.
%BADEXECUTED
Fault events executed.

%BADSCHEC
Fault events scheduled.

%CNT_EXE_CNT
Control events executed.

%CNT_SCH_CNT
Control events scheduled.

%CPU_TIME
Displays the cumulative CPU time used since the process was created. The
time is in milliseconds with a resolution of 10 milliseconds. Since the
cumulative CPU time is reported by VMS system service, it includes the
CPU time for commands, such as SHOW SYSTEM, that are not originated
by the process.

%DETECTION
Faults detected.

%DROP_PASSES
Number of time the fault drop logic was invoked.

%EDIT
The DECSIM edit number. For example, for Version 5.4-1410, %EDIT
returns 1410.

%ELEMENTS
The number of elements in a network. This is 0 if the network is not
loaded.

%EXT_EXE_CNT
Deposit or pattern events executed.

%EXT_SCH_CNT
Deposit or pattern events scheduled.

%FAULTCONVERGE
Fault effects converged

%FAULTDIVERGE
Fault effects diverged.

%FAULTDROPS
Faults detected and dropped.

%FAULTSOURCES
Faults inserted.

%FIRM_DETECTED
Number of firm fault detections made.
%GOODCANCEL
Events scheduled but canceled.

%GOODEXECUTED
Events executed.

%GOODSCHED
Events scheduled.

%INDIRECT_LINE
File line number of an executing indirect file. Use from the SIM* prompt.

%INTERNALTIME
Time in one-half resolution units.

%NODES
The number of nodes in the network. This is 0 if the network is not loaded.

%PAGEFAULTS
Displays count of page faults.

%POSS_DETECTED
Number of possible fault detections made.

%REVISION
The network revision number. If a network is not loaded, the value is 0.

%ROUTINEVALUE
The value of the last routine executed by the CALL command that returned a value. This is updated by the time a WATCH/RETURN is triggered so that %ROUTINEVALUE can be accessed in a WATCH/RETURN do action.

%SCALAR_DROPS
Gate/MOS fault effects deleted by fault drop logic.

%TIME
The current simulation time.

%UNDEFINED
The number of "U"s, or undefined states, in a network. This is 0 if the network is not loaded.

%V_BAD_EXE_CNT
Vector fault events executed.

%V_GOOD_EXE_CNT
Vector events executed.

%V_LIST_CNT
Vertical list count.
DECSIM Environment and Conventions

%VECTOR_DROPS
Vector fault effect deleted by fault drop logic.

%VERSION
The DECSIM version number. For example, for Version 5.4–1410,
%VERSION returns 5.

%VIRTUAL_PEAK
Displays the peak virtual address size.

%WHEEL_HORZ_LEN
Internal time increment (used by developers).

%WS_PEAK
Displays the peak working set size.

%WS_QUOTA
Displays the current working set size quota.

%WS_SIZE
Displays the current working set size.

Note: If a system variable is used in a WATCH or TRACE with
the qualifier /WHEN, /WHILE, or /UNTIL, the system variable is treated
as a constant whose value is captured once before loop execution
begins. System variables cannot be subscripted, except through
the use of the subscript operator, as in: (%system variable)<i>.

The following system variables are ZYCAD-specific and are produced by
replaying the entire .ZBR file. They can be displayed with EVALUATE or
PRINT.

%Z_ERROR_CODE
Highlights ZYCAD errors – call the DECSIM/ZYCAD support team.

%Z_ERROR_SUB_CODE
Highlights ZYCAD errors – call the DECSIM/ZYCAD support team.

%Z_EVENTS_EXECUTED
Displays the total number of events executed by the ZYCAD engine.

%Z_FANOUT_STORAGE
Displays the amount of memory used in the S-modules for network
connectivity (fanout).

%Z_MEMORY_ERRORS
Displays the number of memory errors detected by the accelerator during
simulation. This is incremented during simulation. See Section 1.3.4 for
more information on memory errors.

%Z_MODULES_DOWN
Displays the number of disabled s-boxes. If this number is not zero, call
the DECSIM/ZYCAD support team.
%Z_REPLAY_COUNT
Displays the number of replay events (discrete times in simulation units) that were scheduled on the time wheel. This is incremented during simulation.

%ZRESOURCE_STORAGE
Displays the amount of memory used in the M-module for functional

%Z_RUNTIME
Displays the length of the hard ZYCAD simulation run.

%Z_S1_EXECUTED
Displays the number of events executed by module S1.

%Z_S2_EXECUTED
Displays the number of events executed by module S2.

%Z_S3_EXECUTED
Displays the number of events executed by module S3.

%Z_S4_EXECUTED
Displays the number of events executed by module S4.

%Z_S5_EXECUTED
Displays the number of events executed by module S5.

%Z_S6_EXECUTED
Displays the number of events executed by module S6.

%Z_S6_EXECUTED
Displays the number of events executed by module S7.

%Z_S8_EXECUTED
Displays the number of events executed by module S8.

%Z_S9_EXECUTED
Displays the number of events executed by module S9.

%Z_S10_EXECUTED
Displays the number of events executed by module S10.

%Z_S11_EXECUTED
Displays the number of events executed by module S11.

%Z_S12_EXECUTED
Displays the number of events executed by module S12.

%Z_S13_EXECUTED
Displays the number of events executed by module S13.

%Z_S14_EXECUTED
Displays the number of events executed by module S14.
DECSIM Environment and Conventions

%Z_S15_EXECUTED
Displays the number of events executed by module S15.

%Z_S16_EXECUTED
Displays the number of events executed by module S16.

%Z_SEQ_ERROR
Highlights ZYCAD errors – call the DECSIM/ZYCAD support team.

%Z_TOTAL_GATES
Displays the total number of ZYCAD gates used for the simulation.
memory, patterns, traces, and deposits.

4.2.6.4 State Variables Defined by the STATE Command
Expression operands can also be variables as declared in the STATE
command. These variables normally have bit subscripts and can optionally
be arrays of words and thus have word subscripts. If subscripts are
declared in the STATE command and omitted in expressions, DECSIM
issues a warning message.

If either word or bit subscripts are out of range, an error is generated and
a 0 value is assumed.

4.2.6.5 Subexpressions
Another expression that is used as an operand is called a subexpression.
The order of evaluation depends on operator precedence and parenthesis
placement.

4.2.6.6 Expression Operators
Expressions can have one or more operators to operate on the operand
value. Operators are characterized by the following:

- Function, including 4-state truth table
- Width of result
- One operand (monadic or unary) or two operands (dyadic or binary)
- Precedence

See Appendix D for more detailed information on operators.

4.3 Accessing Symbols in DECSIM
DECSIM allows you to access the different symbols that can appear in
your network description. These symbol types are:

- Instance labels
- Signal names
- Behavior global states
- Behavior local states
- Behavior routine names
- Behavior named constants

4–20
DECSIM Environment and Conventions

- Behavior synonyms
- Behavior ports
- Behavior routine formal parameter names
- Memory array names

DECSIM also allows you to access command states defined in the DECSIM run-time system (not in your network). The command state names should not duplicate the network symbol names. If you do have command states and network symbols that have the same name, the command state name is chosen. However, if you supply the full hierarchical name of the network symbol, then this is chosen.

Symbol names can be abbreviated if you give an unambiguous abbreviation. However, command STATE names cannot be abbreviated.

4.3.1 Access to Unnamed Signals

There are several cases when DECSIM generates unnamed signals:

- A primary input to a MOSFET stack, which has a signal connected to power at the opposite end of the stack.
- A primary input that is generated via a signal declared to be UNCONNECTED_OK.
- A primary input that is given only a label.pin name in the network description.
- A primary input that is declared to be a pullup or pulldown and therefore drives a phantom model.

DECSIM gives these signals a unique "fake" name in the form "$$n\), where n is an integer. You can use this name to access the previously unnamed signal.

Example

```
SIM> EXAMINE $$0
SIM> DEPOSIT $$2 = 1
```

4.3.2 Access Lists in DECSIM Commands

Many DECSIM commands have a list of symbols that acts as the target of the command. This list is called an access list.

Example

```
TRACE A, B, C
PATTERN I1.I2.I1, SNET.L1.IN
EXAMINE A, B<31:0>, C[0:10]
```

Different commands allow different symbols to appear in their access lists, depending on the functionality of the specific command. For example:

- The PATTERN command allows only signal names, since it only drives signals.
4.3.3 Hierarchical Access to Symbols

DECSIM networks are constructed hierarchically. This means that symbols can appear at any level in your network description. Furthermore, the same symbol name can appear in different levels of the hierarchy, but represent different symbols.

Therefore, DECSIM lets you specify a "label path" that identifies the hierarchical path of instance labels leading to a specific symbol.

The general format is a set of labels, separated by dots, followed by the symbol name. The labels, reading left to right, are ordered from the highest hierarchical level to the lowest.

Example

L1:

L2: R A 1 B 2 C
L3: S A 3 B 4 C T Y

- L1, L2, and L3 are instance labels.
- The gates are numbered 1, 2, 3, and 4.

<table>
<thead>
<tr>
<th>Gate</th>
<th>Full Hierarchical Name of the Output Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L1.L2.B</td>
</tr>
<tr>
<td>2</td>
<td>L1.L2.C</td>
</tr>
<tr>
<td>3</td>
<td>L1.L3.B</td>
</tr>
<tr>
<td>4</td>
<td>L1.L3.C</td>
</tr>
</tbody>
</table>

The outputs of gates #1 and #3 have the same signal name ("B"), but are distinguished by their different label paths: "L1.L2" as compared to "L1.L3".

Signals can also be accessed using pin names. See Section 3.3.1.2.2 for more information.
4.3.4 **SET SCOPE Command**

The SET SCOPE command moves your observation point (scope) up or down in the network hierarchy. It allows you to access symbols that fall within the current scope without specifying the full hierarchical name.

In the example above, if you give the "SET SCOPE L1.L2" command, you can access the "B" symbol directly, without having to access it as "L1.L2.B"; DECSIM automatically assumes the label path that leads down the the current scope.

See the SET SCOPE command in Chapter 5 for more information.

4.3.5 **The "%NET" Pseudo-label**

"%NET" is a special label name that means “the top level of the network”. This allows you to supply a full hierarchical name for a signal, even when the current scope is set to a lower-level model. Then you can access symbols that are outside of the current scope.

4.3.6 **Accessing Symbols Using Wildcard Characters**

A symbol name wildcard character is a single character that matches any other character or characters of the symbol name in the corresponding position. The question mark wildcard (?) matches any single character, and the asterisk wildcard (*) matches zero or more characters.

Wildcard characters can be used to match any symbol type: signals, behavior states, etc.

The question mark (?) and asterisk (*) characters are always interpreted as wildcards, and the dot (.) character is always interpreted as a separator, even when they are enclosed in quotation marks. As a result, neither the period (.) nor the wildcard characters should be used in a symbol name.

A wildcard character searches for a match in a single hierarchical level. The string "*" matches all symbols at the current hierarchical level; it does not match all symbols at the current and all lower hierarchical levels.

Wildcard characters can appear many times in any position in the symbol name. Wildcard characters can be used in any level of a hierarchical symbol name; for example, "*.*.*" is valid. In a hierarchical symbol name, both the "label part" part and the "symbol" part can be wildcards. A maximum of 8 wildcard characters can appear in a single hierarchical name.

The commands that allow wildcard characters in their access lists are: DEPOSIT, FAULT, EXAMINE, WATCH, and TRACE. Wildcard characters cannot appear in symbols in expressions.
Example

The examples below show the results of using wildcards in the EXAMINE command with six signals in the network: WABCY, WABCZ, WQQQY, WQQQZ, XABCY, XQQQY.

```
SIM> EXAMINE *                      ! * matches all names
   WABCY = U
   WABCZ = U
   WQQQY = U
   WQQQZ = U
   XABCY = U
   XQQQY = U

SIM> EXAMINE ?Y                      ! * positioned at the front of the name
   WABCY = U
   WQQQY = U
   XABCY = U
   XQQQY = U

SIM> EXAMINE W?Y                     ! * positioned in the middle of the name
   WABCY = U
   WQQQY = U

SIM> EXAMINE W*                      ! * positioned at the end of the name
   WABCY = U
   WABCZ = U
   WQQQY = U
   WQQQZ = U

SIM> EXAMINE ?ABCY                   ! ? positioned at the front of the name
   WABCY = U
   XABCY = U

SIM> EXAMINE WA?CY                   ! ? positioned in the middle of the name
   WABCY = U

SIM> EXAMINE WABC?                   ! ? positioned at the end of the name
   WABCY = U
   WABCZ = U

SIM> EXAMINE *B*                     ! Multiple *'s are valid
   WABCY = U
   WABCZ = U
   XABCY = U

SIM> EXAMINE *A*C*                   ! Multiple *'s are valid
   WABCY = U
   WABCZ = U
   XABCY = U

SIM> EXAMINE *BC?                    ! Mixed *'s and ?'s are valid
   WABCY = U
   WABCZ = U
   XABCY = U
```

Wildcarding in a Hierarchical Name

Assume a network has two labels at its top level: LABELA1 and LABELA2. LABELA1 instantiates a model that has two labels itself: LABELC1, LABELC2. These labels represent models that have signals SIGNALX1, SIGNALX2, SIGNALY1, and SIGNALY2.

LABELA2 instantiates a model that has two labels itself: LABELD1, LABELD2. These labels represent models that have signals SIGNALX1, SIGNALX2, SIGNALY1, and SIGNALY2, also.
The examples below show the results from using wildcards with the EXAMINE command:

SIM> EXAMINE *.** ! All labels and all signals
       LABEL A1.LABEL_C1.SIGNAL_X1 = U ! are wildcarded.
       LABEL A1.LABEL_C1.SIGNAL_X2 = U
       LABEL A1.LABEL_C2.SIGNAL_Y1 = U
       LABEL A1.LABEL_C2.SIGNAL_Y2 = U
       LABEL A2.LABEL_D1.SIGNAL_X1 = U
       LABEL A2.LABEL_D1.SIGNAL_X2 = U
       LABEL A2.LABEL_D2.SIGNAL_Y1 = U
       LABEL A2.LABEL_D2.SIGNAL_Y2 = U

SIM> EXAMINE LABEL A1.** ! Top-level label is supplied;
       LABEL A1.LABEL_C1.SIGNAL_X1 = U ! all other level labels and
       LABEL A1.LABEL_C1.SIGNAL_X2 = U ! signals are wildcarded.
       LABEL A1.LABEL_C2.SIGNAL_Y1 = U
       LABEL A1.LABEL_C2.SIGNAL_Y2 = U

SIM> EXAMINE *.LABEL_C1.* ! Middle-level label is supplied;
       LABEL A1.LABEL_C1.SIGNAL_X1 = U ! all other level labels and
       LABEL A1.LABEL_C1.SIGNAL_X2 = U ! signals are wildcarded.

SIM> EXAMINE *.**.SIGNAL_X1 ! Bottom-level signal name is
       LABEL A1.LABEL_C1.SIGNAL_X1 = U ! supplied; all other level
       LABEL A2.LABEL_D1.SIGNAL_X1 = U ! labels are wildcarded.

### 4.3.7 Specific Symbol Types

#### 4.3.7.1 Instance Labels

Instance labels appear on instance statements in a network description. If a label is not specified on an instance statement, a label of the form "$0", "$1", etc. is automatically generated.

Instance labels specify a "label path" as part of a hierarchical symbol name and specify scope in the SET SCOPE command.

A label name must be enclosed in double quotation marks if it contains a special character (including spaces), or a wildcard character.

Labels are limited to 123 characters.

#### 4.3.7.2 Signal Names

Signal names specify interconnection between elements. They include both actual signal names in an instance statement and model header signals. They also include behavior port names.

A signal name must be enclosed in double quotation marks if it contains a special character (including spaces), or a wildcard character.

Signal names are limited to 31 characters.

Signal names can have bit subscripts of the form <0>, <31:0>, etc. Bit subscripts are optional for signal names that appear in a network description; however, behavior port signal names always have bit subscripts. In the DECSIM command language, if you want to access the full width of a subscripted signal, you can omit the bit subscripts.

The example below assumes that the signal SIG1<31:0> is defined in the network.
Example

EXAMINE SIG1<31:0>  ! Examines the full width of the signal
EXAMINE SIG1       ! Examines the full width of the signal
EXAMINE SIG1<15:0> ! Examines the lower 16 bits of the signal

4.3.7.3 Specifying Assertion and Polarity
This section explains the conventions for naming signals that are asserted low and the conventions for indicating polarity.

Syntax

assertion signal_name polarity

where assertion can be a minus sign (−) or a blank.
where polarity can be one of the following: H, L, 0H, 0L, 1H, 1L, (0)H, (0)L, (1)H, (1)L

A minus sign before a signal name indicates that the signal is asserted low; a blank indicates that the signal is asserted high.

There are a number of assertion and polarity combinations that are equivalent, called synonyms. The list below shows the sets of synonymous forms; the normalized form is listed first in each set.

• signal_name H and −signal_name L
• signal_name L and −signal_name H
• signal_name 0H, signal_name (0)H, signal_name 1L, signal_name (1)L,
  −signal_name 1H, −signal_name (1)H, −signal_name 0L, and −signal_name (0)L
• signal_name 1H, signal_name (1)H, signal_name 0L, signal_name (0)L,
  −signal_name 0H, −signal_name (0)H, −signal_name 1L, and −signal_name (1)L

DECSIM treats the first assertion and polarity specification for a signal name as the primary name. Subsequent forms of the signal name are treated as secondary synonyms. NETPRO enters the normalized form of a signal as a synonym if you specify a nonnormalized form; this gives you access to both the normalized and nonnormalized forms of the signal from the Interactive Command Language.
4.3.7.4 Signal Names in a Hierarchy
The same output or input can have many different hierarchical names.

Example

- L1, L2, and L3 are instance labels.
- The gates are numbered 1, 2, 3, and 4.

In this example:
- The output of Gate #4 can be accessed by the names "Y", "L1.T", and "L1.L3.C".
- Likewise, the output of Gate #2 can be accessed by the names "L1.S", "L1.L2.C", and "L1.L3.A".

DECSIM must decide which of the various hierarchical names is the "preferred name," that is, the name that it displays. The general rules are:

1. For outputs, DECSIM uses the name at the highest level in the hierarchy. In the example below, this means that, for scope at %NET:
   - The preferred name of the output of Gate #4 is "Y".
   - The preferred name of the output of Gate #2 is "L1.S".

For scope at L1.L3:
- The preferred name of the output of Gate #4 is "C".
- The preferred name of the output of Gate #2 is "%NET.L1.S".

2. For inputs, DECSIM uses the name of the input at the next lower level from the current scope. Thus, in the example above, in the scope "L1", the preferred name of the input of Gate #1 is "L2.A", not "R".

Preferred names make it possible to use the EXAMINE command for one signal name and have it display the value of another signal name. By default, the EXAMINE command displays the preferred name; however, the display can be controlled with the /FULL and /EXACT qualifiers.

The following example is based upon the example given above.
DECSIM Environment and Conventions

Example

SIM> EXAMINE L1.L3.C
Y = 1
!
SIM> EXAMINE/FULL L1.L3.C
L1.L3.C = 1
!

4.3.7.5 Behavior Global States

Behavior global states are storage states that are statically allocated in a behavior model. A behavior global state name must be enclosed in double quotation marks if it contains a special character or a wildcard character.

Global state names are limited to 31 characters.

Behavior global state names always have bit subscripts of the form <0>, <31:0>, etc. States can also be defined as array storage, and defined and accessed with word subscripts (that is, enclosed in square brackets). In the DECSIM command language, to access the full width of a subscripted behavior global state, you can omit the bit subscripts.

Behavior global states can be accessed as symbols in the same way as you access signals—they can be arguments of EXAMINE, EVALUATE, TRACE, WATCH commands, etc., and they can be accessed with a full hierarchical name that includes a label path.

Example

L1.L2.S<31:0>
%NET.T1.T2.ARR[0:10]<15:0>

4.3.7.6 Behavior Local States

Behavior local states are storage states in behavior models that are defined inside behavior routines. They are dynamically allocated—this means that they are allocated on a per-activation basis. A behavior local state name must be enclosed in double quotation marks if it contains a special character or a wildcard character.

Behavior local state names are limited to 31 characters. They always have bit subscripts of the form <0>, <31:0>, etc. States can also be defined as array storage, defined and accessed with word subscripts (that is, enclosed in square brackets). In the DECSIM command language, to access the full width of a subscripted behavior local state, you can omit the bit subscripts.

Behavior local states are only accessible when the routine is running. If they are not declared static, they will be reinitialized every time the routine wakes up.

Behavior local states can be accessed as symbols in the same way as you access signals—they can be the target of EXAMINE, EVALUATE, TRACE, WATCH commands, etc., and they can be accessed with a full hierarchical name that includes a label path. To access a behavior local state with a label path, the routine name must appear as the last (right–most) label in the label path.

Example

L1.L2.R1.S<31:0>  ! "L1" and "L2" are instance labels. "R1" is a ! routine name. "S<31:0>" is the STATE name.
%NET.T1.T2.R2.ARR[0:10]<15:0>  !"T1" and "T2" are instance labels. !"R2" is a routine name.
!"ARR[0:10]<15:0>" is the STATE name.
4.3.7.7 Behavior Routine Names
As a symbol type, behavior routine names share the following similarities with instance labels:

- Within the scope of a behavior model, the command EXAMINE/LABELS "*" displays the names of all of the routines defined in that behavior model.
- You can use the command SET SCOPE routine-name to set your scope inside a behavior routine.
- Likewise, a routine name can appear as the lowest-level (right-most) label in a hierarchical label path. This enables you to reference behavior local states that are defined inside the routine.

Behavior routine names in the ACTIVATE and CALL commands can also specify which routines are to be activated or called.

4.3.7.8 Behavior Routine Formal Parameter Names
Behavior routine formal parameter names are treated the same way as the behavior local states.

4.3.7.9 Memory Array Names
Memory array names are those names specified in a memory primitive definition by the ARRAY_NAME attribute. A memory array name must be enclosed in double quotation marks if it contains a special character (including spaces), or a wildcard character.

Memory array names are limited to 31 characters.

Memory array names always have bit subscripts of the form <0>, <31:0>, etc., and also have word subscripts enclosed in square brackets.

Memory array names can be accessed as symbols just as you would access signals—they can be the target of EXAMINE, EVALUATE, TRACE, WATCH commands, etc., and they can be accessed with a full hierarchical name that includes a label path.

Like signals, a memory array can be instantiated many times yet maintain its own distinct memory array allocation area, as designated by the full hierarchical name that includes a label path.

4.3.7.10 Command States
Command states are variables (storage) that are declared in the command language with the STATE command. They can be displayed by the SHOW STATE command and deleted by the CANCEL STATE command.

Command state names cannot be abbreviated. Command states do not have hierarchical names—label paths cannot be used with command states. A command state name must be enclosed in double quotation marks if it contains a special character.

Command state names can have bit subscripts of the form <0>, <31:0>, etc. If you define a command state with no bit subscripts, it is assumed to be 32 bits wide, with bit subscripts <31:0>. Command states can also be defined as array storage, defined and accessed by using word subscripts (enclosed in square brackets).
Command states can be accessed only where expressions are valid—they can appear in the EVALUATE command, for example, but they cannot appear in an EXAMINE command’s access list, and they cannot be watched or traced.

4.3.8 Machine Descriptors

Machine descriptors are optional directives used with signal names that allow you to access faulted machines that are concurrently simulated in the same way as you would manipulate the fault-free (good) simulation.

**Format**

```plaintext
name [ ( machine descriptor [ , ... ] ) ]

where name is:

    signal_ name

where machine descriptor is:

    PRIMARY       ! for the good machine
    type : signal_name  ! for the faulty machine

where type is one or more of the following:

    SA0           ! stuck at 0
    SA1           ! stuck at 1
    ISA0          ! input stuck at 0
    ISA1          ! input stuck at 1
    OSA0          ! output stuck at 0
    OSA1          ! output stuck at 1
```

**Description**

A machine is the good or the faulty concurrent elements that represent internal states during a concurrent simulation. Machine descriptors enhance access to various types of machines for a particular signal.

Machine descriptors can only be used in conjunction with a CONTEXT command, which controls the machine to which the DECSIM commands apply. Potentially, any command that uses an access list can be enhanced with the use of machine descriptors.

A machine descriptor is a signal name modifier that describes and selects the particular concurrent machine.

The possibilities are:

- PRIMARY – The NETPRO compiled machine, sometimes called the good machine. It describes the network defined by the NETPRO wirelist.
- SA0, SA1 – Concurrent (faulted) machines. These specify “stuck at 0,1”.
- ISA0, ISA1 – Concurrent (faulted) machines. These specify “input stuck at 0,1”.
- OSA0, OSA1 – Concurrent (faulted) machines. These specify “output stuck at 0,1”.

4–30
The SA0 or SA1 machines refer to both stuck inputs and outputs, except when a signal name unambiguously defines an input or output such as e1.1, where pin 1 is an input. You can use OSA0 or OSA1 to refer to the machine containing the stuck output of a device or the driver of a bidirectional pin.

Similarly, you can use ISA0 or ISA1 to refer to the machine containing the stuck input of a device or the receiver of a bidirectional pin. If you use ISA0 or ISA1 on an output, or OSA0 or OSA1 on an input, then DECSIM prints an error message.

See also: CONTEXT in Chapter 5.

Examples

**Stuck at Faults**

{SA1:e1.3} Refers to a machine containing a stuck at 1 fault on pin 3 of e1 regardless of whether pin 3 is an input or output. In the case of bidirectional pins, the fault would have at least two sources installed, one for each output and input within e1 connected to the pin.

{ISA1:e1.3} Refers to a machine containing faults on e1.3, which must be an input or bidirectional pin. If e1.3 is an output, an error message appears. In the case of a bidirectional pin, the fault exists on the receivers on that pin within e1.

{OSA1:data0} Refers to a machine containing a fault on the output of the node. If there is only one driver, the fault is on the output of the driving device. If the node has several drivers and is logically OR'd or AND'd together with a phantom, then the faults are on the output of the phantom gate.

All machine descriptor machine types can be abbreviated to the shortest unique string. **Wildcards in Machine Descriptors**

In certain contexts concerning machine descriptors, the wildcard form is useful. The following are examples of valid and invalid uses.

{} refers to the default descriptor (see CONTEXT command)

{~} refers to all concurrent models

{~: } invalid because not all elements of machine types are compatible

{Machine_type_list:*} valid only for elements that are signal names
Examples of list and wildcard use:

\{(SA1,SA0): clk\}  report or access two faulty models: One has stuck at 0 fault source on clk, the other has stuck at 1 fault source on clk.

\{(SA1,SA0):*\}  report or access any machine defined by a stuck at 1 or 0 fault on the signals defined at the top modeling level.

\{SA1:{*,*,*}\}  report or access any machine defined by a stuck at 1 fault on the signals defined at the top and second modeling levels.

4.4 Interactive Features

These features facilitate using DECSIM interactively.

4.4.1 Command Modes and Command Prompting

The DECSIM command mode indicates when it is ready for and expecting terminal input by displaying a prompt at the left margin. There are many modes of input in the command language, each with its own unique prompt. These prompts are shown below.

SIM>
The first prompt you see when running DECSIM. Any command or command macro can be typed at this top-level prompt.

MORE>
Signals that DECSIM is expecting more input, and that you must finish the current command being typed. Some causes of the MORE> prompt could be incomplete command syntax or the use of a line continuation character.

QUOTE>
Indicates that a text string preceded by quotation marks needs to be closed with ending quotation marks.

MACRO>
Appears during macro definition entry (ended by $endmacro).

@XXX>
Indicates that input is being executed from indirect file xxx and, due to the ECHO qualifier, is being echoed on the terminal.

XXX->
where “xxx” is any prompt listed here.

Indicates that, due to a PAUSE or CTRL/C command, input from an indirect file has been temporarily suspended and commands can be input from the terminal. Input can be returned to the indirect file by typing CONTINUE.
CTRL-C>
Occurs after you have typed CTRL/C for an emergency halt. Type H for help.

SIMBREAK>
Denotes a special debug mode. To exit, type CANCEL BREAK.

PATTERNNAME>
Appears when you specify SYS$INPUT as the /FILE in a pattern command. The pattern name appearing as the prompt is the name specified with the /PATTERNNAME qualifier. DECSIM assigns unnamed patterns as PATTERNX. To end the pattern from the pattern prompt, type CTRL/Z. To end this pattern from the SIM> prompt, type CANCEL PATTERN.

4.4.2 Escape Prompting

Escape prompting provides you with interactive help while you are typing commands. If you need help with syntax, or if you want a list of possible entries, press the TAB key. On VMS V3, you can use the escape (ESC) key instead of TAB.

At the prompt

- Pressing TAB once displays the expected syntax.
- Pressing TAB twice displays a list of possible entries.

When you have typed one or more letters of an entry:

- If what you typed is invalid, pressing TAB once produces an error message and prompts you to retype that part of the command following the error.
- If what you typed is ambiguous, pressing TAB once produces a beep; pressing TAB twice displays a list of valid entries.
- If what you typed is a complete field, pressing TAB prompts you for the next field.

Examples

SIM> TAB@ One of the following: !first TAB at the prompt
SIM> @ keyword LABEL
SIM> TAB The legal keywords are: !2nd TAB at the prompt

ACTIVATE ALLOCATE BEGIN BREAK CALL CANCEL
COMMENT COMPILE CONFIGURE CONTEXT DCL DEBUG
*DEPOSIT DETECT DISABLE DROP EDIT ENABLE
ERROR EVALUATE *EXAMINE EXIT FAULT FOR
FORMATT GRIPF HALT HAZARD HELP IF
INDIRECT KEY LEAVE LOAD LOG MACRO
*MODIFY MOS PATTERN PRINT QUIT RADIX
RECALL REPLAY RESTORE SAVE SCOPE SELECT
SET SH SHARE SHOW *SIMULATE SPAWN
STACK STATE STATISTICS SYNONYM TERMINAL *TIME
TIMING TRACE UNLOAD UNTIL USE WATCH
WHILE ZYCAD
Restrictions and Anomalies

The following are known restrictions for escape prompting:

1. Escape prompting does not support user--symbol lookup nor provide lists of file names, times, values, or access lists.

2. Using escape prompting after pressing CTRL/C gives you an error message.

3. Used with the COMPILe command, escape prompting does not prompt you for network descriptions, but simply requests a file name.

4. Used in defining macros, escape prompting does not prompt you for the contents of the macro definition body, but simply requests that you complete the macro text definition.

5. Because certain macro expansion parameters are used in escape prompting, you can use only a restricted set of parameters when defining macros. See MACRO in Chapter 5 for more information.

6. Escape prompting does not handle comments. Escape prompting ignores everything from an exclamation mark (!) to the end of the line.

7. Escape prompting does not check for conflicting or duplicate qualifier names. Escape prompting offers you choices of qualifiers that include any qualifiers you have typed.

4.4.3 CTRL/G

Control G (CTRL/G) brings you back to top-level command mode and aborts all levels of looping. You can use CTRL/G after a syntax error and retype the entire command.

CTRL/G does not cancel an in--progress expansion of a macro or execution of an indirect file. CTRL/G is not recognized if found within an indirect file or macro definition.

4.4.4 CTRL/C Emergency Interrupt

CTRL/C provides emergency halt during simulation execution or command mode. You can use the CTRL/C special command at any time during simulation, in command mode, or during terminal printout.
DECSIM Environment and Conventions

Format
CTRL/C

Options
The following prompt appears when you press CTRL/C:

Yes? (H for help) CTRL–C>

At the prompt, type the letter representing one of the following:

B break
Invokes an interactive debug session (acts as break command). To get back to the CTRL–C> prompt, type CANCEL BREAK.

C continue
Continues operation from the point at which it was interrupted.

D dump
Forces DECSIM to create a DECSIM.DMP file. This option is mainly intended for developers and maintainers of DECSIM.

E exit
Has the same effect as the exit command.

H help
Prints the option list for CTRL/C.

N return next
Finishes executing the current command (whether entered interactively or from an indirect file), and then returns to command mode before executing the next command.

Q quit
Same as the quit command.

R return
Finishes the current time frame of simulation, then returns to command mode.

T time
Invokes the DECSIM time command, then returns you to the CTRL/C prompt.

Note: Use PAUSE, not CTRL/C, to suspend input from an indirect file or macro definition.

If the input stream was from an indirect command file, the indirect file is temporarily suspended, and the terminal is in a command mode prompted by SIM*>. Most commands can normally be invoked from this mode, and the indirect file execution can be continued (with the CONTINUE built-in command macro) or canceled (with the CANCEL INDIRECT command). If a SIMULATE command was in progress from an indirect file, typing SIMULATE continues the interrupted SIMULATE command.

Caution: Two consecutive CTRL/C’s terminate your session immediately.
4.4.5 Break Points

By using the BREAK command in conjunction with the WATCH or TRACE command before simulating, you can pause in the middle of the simulation. You then can use any command except SIMULATE. For behavior models, once you have interrupted the simulation, you can use the STEP and RESUME commands either to step through the code or to resume the simulation.

4.5 Specialized Uses of DECSIM

The following three sections provide a brief introduction to fault simulation and hazard analysis in DECSIM and to the ZYCAD hardware accelerator.

4.5.1 Fault Simulation

In DECSIM, fault simulation is concurrent. After you attach a fault source to a node, DECSIM simulates the behavior of a good, or unfaulted network, and then fault simulates only those paths where a fault has caused a deviation from the good network. If, downstream from the fault source, the faulty network becomes the same as the good network, DECSIM stops simulating that path.

You can specify that a certain signal, list of signals, or category of signal such as all inputs, be stuck at a certain logic state. DECSIM inserts the faults, then uses concurrent simulation to calculate the differences from the good network.

To make simulation run as fast as possible, DECSIM collapses equivalent faults. At fault insertion time, DECSIM classifies equivalent faults into a single class and inserts only one fault from each class. For example, if the input of an AND gate is stuck at 0 and the output of the same AND gate is also stuck at 0, DECSIM classifies these two faults as equivalent and inserts just the fault on the output. Fault collapsing does not affect fault detection, however. When a fault is detected, DECSIM reports all the faults in that class as detected.

You can insert faults in equation, memory, and MOS models, but not in behavior or RealChip models.

The ability to simulate many faulted networks simultaneously allows the engineer:

- To evaluate a given test pattern or diagnostic program to establish the percentage detection and the identity of detected and undetected faults before hardware is available.
• To generate a fault dictionary, which gives precise information regarding when and where particular fault symptoms appear at network output points.

Typical applications of DECSIM fault simulation include:

• Interactive creation of test patterns and diagnostic programs.
• Automatic test generation for circuits using heuristic and random test generation techniques.
• Automatic test generation for networks using algorithmic techniques, such as the D-algorithm.
• Small sample fault simulation or sensitive stages simulation to evaluate roughly the effectiveness of a test or diagnostic.

DECSIM's fault simulation capabilities include:

• Stuck at faults
• Fault collapsing to eliminate equivalent faults
• Fault dropping to remove detected faulty networks from simulation

4.5.2 DECSIM and ZYCAD

The ZYCAD Logic Evaluator (LE) and Fault Evaluator (FE) are batch hardware simulation engines that execute approximately one million events per second per S_module when the ZYCAD LE pipeline is kept full. For example, an LE with four S_modules executes approximately four million events per second. The LE acts as a performance improvement tool for DECSIM. ZYCAD applications include design verification, regression testing, and fault simulation.

ZYCAD functionality is a subset of DECSIM functionality. The following section describes some of the ZYCAD restrictions.

Both the LE and the FE simulate only logic models; they do not simulate behavior models or MOS models. The LE does serial fault simulation, whereas the FE does concurrent fault simulation. Both the LE and the FE do good simulation of gate level models.

You cannot define ZYCAD primitive models. However, you can build your own hierarchical models by instantiating the primitive models contained in ZGATES.NET.

ZYCAD memories, both hard (executed on the LE) and soft (executed by DECSIM), use a variation of edge-trigerring called ZYCAD edge-trigerring. Soft DECSIM's default memory simulation is level-sensitive. See Section 1.3.2 for more information on memories.

The ZYCAD LE (hard DECSIM) is not an interactive tool. Therefore, it is not recommended for debugging. You can debug your model using ZYCAD-style soft DECSIM (a version of DECSIM that does not run on the ZYCAD machine, but which duplicates its results).

Most soft DECSIM commands apply to ZYCAD. ZYCAD-specific commands and qualifiers are described with soft DECSIM commands.
You access the ZYCAD Logic Evaluator in a shared environment. Simulating has several steps:

1. You create a network for ZYCAD simulation on your home VAX running DECSIM.

2. You use the SIMULATE command, which creates a binary file (ZIF) and sends it via DECNET to the target VAX, which has the LE attached.

3. The ZIF file is queued for simulation, simulated on the LE, and the binary output results are returned to your home VAX.

4. The report writer produces an ASCII file of the results, and you are notified that the simulation is complete.

The DECSIM/ZYCAD interface uses certain logical names:

- **ZYCAD$NOTIFY** — specifies how you are notified when the ZYCAD simulation run is complete. Your choices are BROADCAST, MAIL, or FILE. FILE creates a .ZER file describing errors. MAIL is the default, and is used when a run has an error.

- **ZYCAD$S_MODULES** — defined at installation time for the entire team. Specifies how many S_modules are in the ZYCAD Logic Evaluator.

- **ZYCAD$NODE** — defined at installation time for the entire team. It specifies the target node, which is defined with no double colons. For example, 'DEFINE ZYCAD$NODE ELSIE'

- **ZYCAD$OUT_STATUS** — normally defined by developers. Defining it as "keep" enables you to keep temporary files that help developers debug the interface.

ZYCAD produces eight files that are listed by their file extensions:

- **ZIF** — ZYCAD input file (your network and stimulus in binary format). The SIMULATE command sends this file to the target node.

- **ZOF** — ZYCAD output file. This is an ASCII file similar in format to a TRACE/VECTOR/ACTIVITY file produced by soft DECSIM. It reports the outcome of your simulation.

- **OUT** — ZYCAD temporary file. This file contains the binary results of the simulation. The report writer uses this information to produce a ZOF file.

- **ZTF** — ZYCAD temporary file. This file tells the report writer how to format the information in the OUT file to produce the ZOF file.

- **ZBA** — ZYCAD binary activity file. This file is the most compact of the ZYCAD files, making it the preferred tool for having signal activity analyzed by another program. See the DECSIM/ZYCAD team for more information on this file.

- **STT** — Shared environment statistics file. This details the time spent to:
  1. Copy the ZIF file to the target node.
  2. Queue for the LE.
  3. Simulate.
4.5.3 Hazard Analysis

A hazard is the possibility that, because of propagation delays in a logic circuit, an output may change briefly to an incorrect state. In Figure 4–1, for example, there is a hazard at the output of the OR gate. If there is a transition on S, the delay caused by the inverter may be enough to cause a brief transition to the wrong state on the output of the OR gate. The pulse may last long enough to set the latch.

Figure 4–1 Logic Circuit with Hazard

When you simulate a model with DECSIM, you can detect this kind of timing violation by setting hazard detection points, hazard checkers, on critical signals in your network. These hazard checkers are usually associated with clocked devices, such as latches, flip-flops, and RAMs.

Because DECSIM's hazard analysis takes place during simulation, it is pattern-dependent. You can detect only those hazards that might occur as a result of the patterns you input to the model.
4.5.3.1 Features
With DECSIM's hazard analysis you have the following options:

- To check for four types of hazards on scalar signals in gate, MOS or memory models:
  - Pulsewidth
  - Cyclewidth
  - Setup
  - Hold

- To define hazard checkers in a model or from the command language

- To define any checker as a conditional checker, activated only when an enable signal is at a specified state

- To define offset in the setup and hold checkers

- To customize the amount of hazard information reported by checkers defined in the command language

- To display information about a checker or a group of checkers during a simulation

- To cancel, enable, or disable a checker or group of checkers during a simulation

4.5.3.2 Restrictions
The following restrictions apply to simulation with hazard checkers:

- Hazard checkers are not allowed on vector signals (behavioral or memory).

- Hazard checkers are ignored in ZYCAD networks and during fault simulation.

Additional restrictions that apply to defining hazard checkers in your model are noted in Section 1.4.1.

4.5.3.3 Hazard Analysis in the Design Process
You may want to implement hazard analysis when you are verifying the design of your structural model.

There are four steps in implementing hazard analysis:

1. Defining the hazard checkers
2. Simulating with hazard checkers
3. Analyzing the violation reports
4. Redesigning your model, if necessary

These steps are described in detail below.
1 Defining the hazard checkers

DECSIM’s hazard checkers are a unique type of simulation artifact. Like watches, detects, and other simulation artifacts that are defined and activated at simulation time, hazard checkers are not part of the permanent network topology; unlike other simulation artifacts, they can be instantiated in structural models.

In DECSIM you can set hazard checkers on scalar signals in structural, memory, or MOS models. You can set checkers when you design your model (NETPRO checkers) or during simulation (command language checkers); you may decide to do both. For information on the syntax of NETPRO hazard checkers, see Section 1.4. For information on the syntax of command language checkers, see the [SET] HAZARD command description in Chapter 5.

In both NETPRO and command language checkers, you define the functionality of a particular hazard checker by passing arguments to the hazard checker parameters and by specifying the signal(s) that the hazard checker will monitor.

The default reporting of NETPRO and command language checkers is the same. You can override these defaults in command language checkers; for example, you can specify that checker successes are reported. You can also specify a full or brief report, or you may limit the report to a particular type of violation. You have the additional option of routing the hazard information of related command language checkers to various log files.

2 Simulating with hazard checkers

You may want to simulate your model first without hazard checkers in order to detect the gross timing errors and logic errors. If you have defined NETPRO checkers in the model, you can cancel or disable them with the CANCEL HAZARD or DISABLE HAZARD command after you compile your model, and then simulate as usual. Redesign the model if necessary.

When you have removed the obvious bugs from your design, then you are ready to simulate with the NETPRO hazard checkers, if there are any in your model. First, execute a [SET] LOG/RESPONSES command to open a log file for the NETPRO hazard information and then compile or load your model. Simulate as usual.

Note: You must execute a [SET] LOG command before loading or compiling your model; otherwise, the NETPRO hazard information will be routed by default to the screen only. Use the /HAZARD qualifier on the [SET] LOG command to put a .HAZ extension on the log file.

If there are no NETPRO checkers in your model, you can define command language hazard checkers in an .IND file and execute the file before simulating your model. If you use the /LOG qualifier in the command language checkers, it identifies (and opens, if necessary) the log file to which the hazard information is routed.
3 Analyzing the violation reports

After you have reviewed the hazard log files from the first simulation, simulate again, using the DISABLE HAZARD or CANCEL HAZARD command to turn off the NETPRO hazard checkers (if necessary) and the [SET] HAZARD command to define new checkers on the same or different signals. The EXAMINE/FANIN command may be useful in tracing the source of the hazard signals.

See Section 4.5.3.4 for a description of the types of violations that can be generated by the hazard checkers.

4 Redesigning your model

When you have identified the sources of the timing violations, you can redesign your model to eliminate them. Choosing faster or slower components or adding redundancy, for example, are ways to eliminate hazards.

Once you have eliminated the hazards, you can cancel or disable the hazard checkers in subsequent simulation and in fault simulation.

4.5.3.4 Hazard Checker Violations and Boundary Conditions

DECSIM's hazard checkers report two types of violations, time and data. You also have the option in command language checkers to specify that the checkers report successes as well. The types of violations that hazard checkers report depend on the type of checker and on which of the optional qualifiers, if any, are specified in the checker. The following sections describe the violations each type of checker will report.

Pulsewidth Checkers

The pulsewidth checker checks the waveform of a repeating signal, such as a clock, during the window of time you specify in the checker definition. A pulsewidth is a change to and from a specified state.

The pulsewidth checker performs either a level check or an edge check, depending on whether you specify a data state or a data edge in the hazard definition. In a level check, the pulse duration at the specified state is compared to the minimum and/or maximum width specifications in the checker definition. If the pulsewidth is shorter than the minimum width or longer than the maximum width, the checker reports a time violation. Data violations never occur in a pulse level check.

In an edge check, the checker reports both time and data violations. For example, when the specified edge is 0→1→0, the checker expects a pulse of 0→1→0. If the pulse waveform beginning with the specified data edge is not square (for example, 0→1→U#2→0), the checker reports a data violation. The checker reports a time violation if the pulsewidth is shorter than the minimum width or longer than the maximum width.

During simulation, the pulsewidth checker starts processing when a transition occurs to the specified data state (level check) or with the specified data edge (edge check). If both a minimum and maximum width of the waveform are specified in the hazard definition, processing continues until one violation occurs, or until both conditions are satisfied. Processing resumes with the next transition to the specified data state or with the specified data edge.
Note: A pulse waveform that is the same duration as the specified minimum width or maximum width does not constitute a violation.

Example: Pulsewidth Level Checker

The following example shows a pulse level checker, first as defined in NETPRO, then as defined in the command language.
No data state is specified, so this checker will monitor the data signal at the default state of 1.

```
%PULSEWIDTH [MIN_WIDTH = 10,
             MAX_WIDTH = 20 ] DATA/clock;
```

```
SIM> SET HAZARD/TYPEx=PULSE
     /MIN_WIDTH=10
     /MAX_WIDTH=20    clock
```

In Figure 4-2 you can see sample results of simulating with this checker. The checker is activated when the clock signal goes to 1 (the default data state) at 10ns. When the signal goes to 0 at 20ns, the checker reports a success because the pulse duration (10ns) is within the specified limits. The checker is activated again at 30ns and reports a time violation at 35ns because the pulse lasted only 5ns. Activated again at 40ns, the checker reports a time violation at 60ns when the pulse duration exceeds the maximum width of 20ns. Note that the checker reports a success on the final pulse, which transitions to U#2 after 10ns.

**Figure 4-2  Pulsewidth Level Checker**

```
  1
  clock 0

SU -- success  \  -- U#2
TV -- time violation  \ -- checker activation

0 35 50 100ns
```
Example: Pulsewidth Edge Checker

The following examples show a pulse edge checker, first as defined in NETPRO, then as defined in the command language.

```
Pulse3: PULSEWIDTH [MIN_WIDTH = 10,
                     MAX_WIDTH = 20,
                     DATA_EDGE = 1_0 ] DATA/strobe0;
```

```
SIM> SET HAZARD/TYP= PULSE
       /MIN_WIDTH=10
       /MAX_WIDTH=20
       /EDGE=DATA=1_0
       /HAZARDNAME=pulse3  strobe0
```

Figure 4–3 illustrates sample results of simulating with this checker. The checker is activated at 20ns when strobe0 transitions from 1 to 0. The checker reports a data violation when the signal goes to U#2 at 30ns instead of going to 1 as expected. At 40ns the checker starts processing again and reports a time violation when strobe0 transitions to 1 at 45ns, violating the minimum width specification. The checker reports a success at 65ns, after strobe0 transitions to 0 at 55ns and back to 1 10ns later. The checker reports a time violation on the final pulse when strobe0 does not transition back to 1 within the maximum width specification.

![Figure 4–3 Pulsewidth Edge Checker](image)
Cyclewidth Checkers

The cyclewidth checker is similar to the pulsewidth checker except that it checks the duration of a cycle, not a pulse, in the waveform of a repeating signal. A cycle is defined as a transition to a specified state, a transition to another state or states, and then a transition back to the specified state.

The cyclewidth checker performs either a level check or an edge check, depending on whether a data state or a data edge is specified in the hazard definition. In a level check, the cycle duration is compared to the minimum and/or maximum width specifications in the checker definition. If the cyclewidth is shorter than the minimum or longer than the maximum, the checker reports a time violation. Data violations never occur in a level check.

In an edge check, the checker reports both time and data violations. For example, when the specified edge is 0→1, the checker expects a cycle of 0→1→0→1. If the waveform beginning with the specified data edge is not square (for example, 0→1→0#2→0→1), the checker reports a data violation. If the cyclewidth is shorter than the minimum or longer than the maximum, the checker reports a time violation.

The cyclewidth checker starts processing when the data signal transitions to the specified data state (level check) or with the specified data edge (edge check). If both a minimum and maximum width of the waveform are specified in the hazard definition, processing continues until one violation occurs, or until both conditions are satisfied. Processing resumes with the next transition to the specified state or with the specified data edge.

Note: A cycle waveform that is the same duration as the specified minimum width or maximum width does not constitute a violation.
Example: Cyclewidth Level Checker

The following examples show a cyclewidth level checker, first as defined in NETPRO, then as defined in the command language.

Example: Cyclewidth Level Checker

Cycle3: %CYCLEWIDTH [MIN_WIDTH = 10, MAX_WIDTH = 20, DATA_STATE= 0 ] DATA/strobe0;

SIM> SET HAZARD/TYPE=CYCLE /MIN_WIDTH=10 /MAX_WIDTH=20 /STATE=DATA=0 /HAZARDNAME=Cycle3 strobe0

Figure 4-4 shows sample results of simulating with this checker. The checker is activated at 10ns when strobe0 transitions to 0, the specified data state. When strobe0 goes back to 0 from 1 at 30ns, meeting the specified minimum/maximum widths, the checker reports a success. The checker starts processing the second cycle waveform at 30ns and reports a time violation when the cycle ends in 8ns, below the minimum of 10ns. The 1_0 transition at 38ns also begins the third cycle waveform. At 58ns, the checker reports a time violation because cycle waveform exceeds the maximum width specification. Note that the checker reports a success on the fourth cycle waveform, because even though strobe0 transitions to U#2, the cycle meets the width requirements.

Figure 4-4 Cyclewidth Level Checker

<table>
<thead>
<tr>
<th>SU</th>
<th>TV</th>
<th>TV</th>
<th>SU</th>
<th>TV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strobe0 0</td>
<td>38</td>
<td>50</td>
<td>58</td>
<td>78</td>
</tr>
</tbody>
</table>

SU -- success  U#2 -- U#2
TV -- time violation  \( \downarrow \) -- checker activation
Example: Cyclewidth Edge Checker

The following examples show a cyclewidth edge checker, first as defined in NETPRO, then as defined in the command language.

```
%CYCLEWIDTH [MAX_WIDTH = 25, DATA_EDGE = 0_1 ] DATA/strobel;
```

```
SIM> SET HAZARD/TYP=CYCLE
   /MAX_WIDTH=25
   /EDGE=DATA=0_1   strobel
```

In Figure 4–5 you can see sample results of simulating with this checker. The checker is activated when the specified 0_1 edge occurs on the strobel signal at 10ns. The cycle waveform that follows the edge is square and meets the width specification, so the checker reports a success at 30ns. The second cycle waveform, which starts at 30ns, ends in a data violation when strobel goes to U#2 at 35ns. The checker is activated again at 60ns when strobel goes from 0 to 1 and reports a time violation when the cycle is not completed within the maximum width specification of 25ns.

**Figure 4–5  Cyclewidth Edge Checker**

```
SU  DV  TV

1

strobe 0

0          35          50          85          100  ns

SU  --  success
DV  --  data violation
TV  --  time violation
□  --  U#2
¶  --  checker activation
```
Setup Checkers

A setup checker defines a minimum time window (the setup window) during which the data signal must be stable before an edge on the reference signal. You have the option of specifying the particular edge(s) on the reference signal and the state(s) at which the signal must be stable.

If you specify an offset for the setup window, any transition that occurs within offset time units before the edge on the reference signal is not reported as a violation. See Figure 4–6 below.

Figure 4–6  Offset in Setup Checkers

![Diagram of setup window and offset]

If you specify data state(s) in the checker definition, the checker reports a data violation if the data signal is not at one of those states during the setup window. The time that the data state is checked depends on whether you specified an offset or not. With no offset specified, the data state is checked at the same time slot as the edge on the reference signal. With an offset specified, the data state is checked at the time slot that is offset time units before the reference edge. If the data signal transitions during the time slot, the last state is the one that is used.

If the data signal does not remain stable during the setup window, the checker reports a time violation. If a sufficient amount of time has not elapsed since the the network was loaded, or since the hazard was defined or enabled (see Conditional Hazards below), the checker reports an insufficient history time violation.

Note: A transition on the data signal at the beginning or end of the setup window does not constitute a hazard violation.
Example: Setup Checker

The following examples illustrate a setup checker with specified data states and reference edge, first as defined in NETPRO, then as defined in the command language.

```plaintext
%SETUP [WINDOW = 10,
    REFERENCES_EDGE = 1_0,
    DATA_STATE = (0,1) ] DATA/ data1, REFERENCE/ clock;

SIM> SET HAZARD/TYPb=SETUP
/REFERENCE=1_0
/STATE=DATA=(0,1)
/REFERENCE=1_0
```

In Figure 4–7 you can see sample results of simulating with this checker. The checker is activated at 10ns, which is start of the specified setup window before the 1_0 edge on the clock signal. Note that although data1 transitions at the beginning and end of this setup window (at 10 and 20ns), the waveform remains stable during the setup window, and the checker reports a success at 20ns. The checker is reactivated at 30ns and reports a data violation at 40ns because data1 is at U#2, which is not one of the specified data states. The checker is activated again at 50 and 70ns and reports time violations at 60 and 80ns because data1 transitioned during the setup windows. The checker reports a success at 100ns, because data1 is stable at 0 for 10ns before the 1_0 edge on the clock signal.

Figure 4–7  Setup Checker

---

**SU** -- success

**DV** -- data violation

**TV** -- time violation

---

**SU** -- success
**DV** -- data violation
**TV** -- time violation

---

4–49
Example: Setup Checker with Offset

The following examples illustrate a setup checker with specified offset, first as defined in NETPRO, then as defined in the command language.

```
Setup3: %SETUP [WINDOW=5,
    DATA_OFFSET=5,
    REFERENCE_EDGE = 1_0,
    DATA_STATE = (0,1)  ] DATA/ data2, REFERENCE/ clock.
```

```
SIM> SET HAZARD/TYPe=SETUP
    /WINDOW=5
    /OFFSET=DATA=5
    /EDGE=REFERENCE=1_0
    /STATE=DATA=(0,1)
    /HAZARDNAME=Setup3
    /REFERENCE=clock  data2
```

The following figure, Figure 4-8, illustrates sample reports from simulation with this checker. Note that this checker is identical to the setup checker in the previous example except for the offset and the window width. The sample waveforms are also identical.

The first two reports from these checkers are the same. However, the setup checker with offset reports a success at 60ns, whereas the setup checker with no offset reports a time violation. No time violation is reported by the offset setup checker because the transition on data2 occurs at 55ns, which is within the specified offset (5s before the 1_0 edge on the reference signal). The next transition on data2, which occurs at 72ns, does trigger a time violation report because it is within the setup window.

**Figure 4-8 Setup Checker with Offset**
Hold Checkers

A hold checker is symmetrically opposite to a setup checker, relative to the reference edge. The hold checker defines a minimum time window during which the data signal must be stable beginning with an edge on the reference signal. You have the option of specifying the particular edge or edges on the reference signal and the state(s) at which the data signal must be stable.

You can also specify an offset for the hold window. If an offset was specified, processing starts offset time units after an edge on the reference signal. If a transition occurs within offset time units after the reference edge, it is not reported as a violation. See Figure 4–9 below.

Figure 4–9 Offset in Hold Checkers

![Diagram](image)

If you specify data state(s) in the checker definition, the checker reports a data violation if the data signal is not at one of those states at the beginning of the hold window. The time that the data state is checked depends on whether you specified an offset or not. With no offset specified, the data state is checked at the same time slot as the edge on the reference signal. With an offset specified, the data state is checked at the time slot offset time units after the reference edge. If the data signal transitions during the time slot, the last state is the one that is used.

If the data signal does not remain stable during the hold window, the checker reports a window violation. If another reference edge occurs before the end of the hold window, the checker reports a reference edge time violation.

Note: If the hold window has no offset, a transition on the data signal at the same time as the edge on the reference signal constitutes a time violation, while a transition at the end of the hold window does not. If the window is offset, a transition on the data signal at the beginning or end of this hold window does not constitute a violation.
Example: Hold Checker

! The following examples illustrate a hold checker with no offset, ! defined first in NETPRO, then in the command language.
.
Hold3: %HOLD [WINDOW = 10000ps,
REFERENCE_EDGE = (0_1, U_1),
DATA_STATE = (0, 1)] DATA/d0, REFERENCE/strobe0;
.
.
SIM> SET HAZARD/TYPE=HOLD
/WINDOW=10000PS
/REFERENCE=strobe0
/EDGE=REFERENCE=(0_1, U_1)
/STATE=DATA=(0, 1)
/HAZARDNAME=Hold3 d0

Figure 4-10 illustrates sample results of simulation with this checker. The checker is first activated at 10ns by the U_1 transition on strobe0. At 19ns, the checker reports a success because d0 is stable at 1 for 10ns (between 10ns and 19ns, inclusive). At 30ns the checker starts processing again because of the 0_1 edge on strobe0. It reports a time violation at 33ns when d0 goes to 0 during the hold window. At 54ns and 79ns, the checker also reports time violations because of transitions on d0 during the hold window. The checker reports a data violation when it is activated at 90ns because d0 is not at 0 or 1.

Figure 4-10  Hold Checker

<table>
<thead>
<tr>
<th></th>
<th>SU</th>
<th>TV</th>
<th>TV</th>
<th>TV</th>
<th>DV</th>
</tr>
</thead>
<tbody>
<tr>
<td>strobe0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d0</td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SU -- success  
DV -- data violation  
TV -- time violation  
□ -- U#2  
▽ -- checker activation
Example: Hold Checker with Offset

The following examples illustrate a hold checker with offset, defined first in NETPRO, then in the command language.

\%HOLD
WINDOW=6,
REFERENCE_EDGE=(0_1, U_1),
DATA_STATE=(1,0),
DATA_OFFSET=4
) DATA/d0, REFERENCE/strobe0;

SIM> SET HAZARD/TYPE=HOLD
WINDOW=6
EDGE=REFERENCE=(0_1, U_1)
REFERENCE=strobe0
STATE=DATA=(0,1)
OFFSET=DATA=4
d0

Figure 4-11 illustrates sample results of simulation with this checker. Except for the window and offset specifications, this checker is identical to the previous hold checker, and the sample waveform is the same. The violation reports are the same, with two exceptions.

The hold checker without offset reports a timing violation at 33ns, whereas the hold checker with offset reports a success at 39ns. In the offset checker, the hold window does not begin until 34ns, and d0 is stable from then until 39ns, thus satisfying the hold window.

Note also that the hold checker with offset does not report the final data violation until 93ns, when the offset ends. The checker without offset reports the data violation at 90ns.

Figure 4-11 Hold Checker with Offset

---

SU -- success
DV -- data violation
TV -- time violation
\( \triangledown \) -- checker activation
Conditional Checkers

A conditional checker is one that is activated only when another signal is at a particular state or set of states. Any of the four types of hazard checkers described above can also be defined as a conditional checker by specifying an enable signal in the checker definition. You also have the option of specifying the state(s) of the enable signal.

The declaration of an enable signal in a hazard definition indicates that the hazard checker is activated only when the enable signal is at a particular state or set of states. The time that the enable signal is checked depends on the type of hazard.

Table 4-1  Enable Signal Check Points

<table>
<thead>
<tr>
<th>Hazard Type</th>
<th>When Enable Signal Is Checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>cyclewidth</td>
<td>first edge of the data signal waveform</td>
</tr>
<tr>
<td>pulsewidth</td>
<td>first edge of the data signal waveform</td>
</tr>
<tr>
<td>setup</td>
<td>edge on the reference signal</td>
</tr>
<tr>
<td>hold</td>
<td>edge on the reference signal</td>
</tr>
</tbody>
</table>

If the enable signal changes at the same time as the data edge (in pulsewidth and cyclewidth checkers) or at the same time as the reference edge (in setup and hold checkers), it may or may not activate the checker. The order in which the signal change events are processed in the given time slot determines whether the checker is activated.

Example

! This hold checker is also a conditional checker,
! because it is activated only when the enable signal is
! at 1 at the time of the _1 _0 edge on the reference signal.
.
.
%HOLD [WINDOW = 10,
    REFERENCE_EDGE = 1 _0,
    ENABLE_STATE = 1,  ] data2, clock, enable
.
.
SIM> SET HAZARD /TYPE=hold
/WINDOW=10
/REFERENCE=clock
/EDGE=REFERENCE=1 _0
/STATE=ENABLE=1
/ENABLE=enable
   data2

Figure 4–12 illustrates a sample simulation with this checker. This checker is first activated at 20ns, when the enable signal is 1 and a _1 _0 edge occurs on the clock signal. Data2 stays stable at 0 from 20 to 29ns, inclusive, so the checker reports a success at 29ns. The checker is activated again at 40ns, but this time data2 transitions to _U2 5ns later, and the checker reports a time violation. Note that although a _1 _0 edge occurs on the clock signal at 60, 80, and 100ns, the enable signal is low and the checker is not activated.
Figure 4-12  Conditional Checker

For information on defining hazard checkers in your model, see Section 1.4. For information on defining hazard checkers during simulation, see the [SET] HAZARD command description in Chapter 5.

See also the following command descriptions in Chapter 5:

[SET] HAZARD
SHOW HAZARD
CANCEL/ENABLE/DISABLE HAZARD
[SET] LOG/HAZARD
Each command description includes some or all of the following information:

- Command function
- Command format
- Command arguments
- Command qualifiers
- Detailed description of command and related information
- Annotated examples

Prefixes are usually listed with the commands with which they occur, not separately. For instance, SET LOG, SHOW LOG, etc. are all described under the LOG command. In some cases, there are separate command descriptions. For example, CANCEL INDIRECT, SHOW INDIRECT, and ENABLE/DISABLE INDIRECT are all described in separate sections immediately following the INDIRECT command.

Optional syntax is enclosed in brackets ( [ ] ).
ACTIVATE

Controls the scheduling of specified behavior routines for later simulation.

**FORMAT**

```
[SET] ACTIVATE   [/qualifier] routine_name
       [(parameter_list)]
SHOW ACTIVATE   [/show_qualifier]
CANCEL ACTIVATE routine_name
```

**Default Qualifiers**

```
/TIME = 0
```

**Default SHOW Qualifiers**

```
/EXECUTE
```

**Optional SHOW Qualifiers**

```
/ACTIVATE = (routine_name [...])
/WAIT
```

**restrictions**

- The ACTIVATE command is used with behavior routines only.

**ARGUMENTS**

- `routine_name`
  Name of a routine. The routine may be prefixed by a sequence of labels that identify its location in the hierarchy. For example, L1.L2.routine_name describes a routine located in model L2 which is located in model L1. You can also use the wildcard ("*") with routine names.

- `parameter_list`
  A list of expressions, separated by commas. The number of expressions must match the number of formal parameters defined in a routine. The value of each expression is placed in the corresponding formal parameter.

**QUALIFIERS**

```
/TIME=]time [units]
```

Schedules the time at which a behavior routine is activated. Time may be relative or absolute. The default is /TIME = 0.

**SHOW QUALIFIERS**

```
/ACTIVATE=(routine_name [...])
```

Shows routines in the access list that are scheduled for activation.

```
/EXECUTE
```

Shows the routines that are currently executing as a result of an activation event. This is the default.

5–2
/WAIT
Shows the routines in the access list that are waiting as a result of the Behavior language WAIT statement.

DESCRIPTION
The SET ACTIVATE command schedules a behavior routine for execution during simulation; the execution of the routine occurs after you enter a SIMULATE command. This differs from the CALL command, which executes a routine from command mode immediately.

The time may be either relative (/TIME=time) or absolute (/TIME= | time). If no time is given, the routine is scheduled for the present time (equivalent to /TIME=0).

CANCEL ACTIVATE removes a routine activation from the event queue. If more than one activation exists for the same routine, the first such activation as reported in SHOW ACTIVATE is deleted (see the example below).

SHOW ACTIVATE/qualifier prints the status of routine activations. Used without a qualifier, SHOW ACTIVATE prints the name of the executing routine.

SHOW ACTIVATE/ACTIVATE reports the model label, routine name, and the status of the activated routines.

Routines in a call stack are shown in reverse order.

Routines activated by another routine are shown with a label, but routines called by another routine are not.

Routines suspended by the WAIT statement are shown with the time they will be reactivated.

For more information see: CALL and SIMULATE

EXAMPLES

SIM> watch/call-btrn do begin show time; sh activate/evaluate/watch="*" -
/activate="*"; end
%I, [SIMWTV] WATCH/TRACE Name is WATCH1
SIM> activate grtn(1)
SIM> activate/time=5 artn
SIM> activate/time=10 artn
SIM>
SIM> show activate/activate="*"

<table>
<thead>
<tr>
<th>Model</th>
<th>Routine</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>ARTN</td>
<td>ACTIVATE 5 nS</td>
</tr>
<tr>
<td>I1</td>
<td>ARTN</td>
<td>ACTIVATE 10 nS</td>
</tr>
<tr>
<td>I1</td>
<td>GRTN</td>
<td>ACTIVATE 10NS</td>
</tr>
</tbody>
</table>
SIM> cancel activate artn
%! [SIMACA] One activation canceled
SIM> show activate/activate="*

    Model  Routine  Status
    I1  ARTN    ACTIVATE |0 nS
    I1  GRTN    ACTIVATE |0 nS

SIM> cancel activate artn
%! [SIMACA] One activation canceled
SIM> show activate/activate="*

    Model  Routine  Status
    I1  GRTN    0  ACTIVATE |0 nS

SIM> sim 5
Simulation Clock= |0 nS
%! [SIMANA] No routines match SHOW ACTIVATE option /ACTIVATE
SIM> show activate/activate="*

    Model  Routine  Status
    I1  BRTN    EXECUTE
    I1  GRTN    WAIT |0 nS
Simulation Clock= |1 nS
%! [SIMANA] No routines match SHOW ACTIVATE option /ACTIVATE
SIM> show activate/activate="*

    Model  Routine  Status
    I1  BRTN    EXECUTE
    I1  GRTN    WAIT |2 nS
Simulation Clock= |2 nS
%! [SIMANA] No routines match SHOW ACTIVATE option /WAIT
%! [SIMANA] No routines match SHOW ACTIVATE option /ACTIVATE
SIM> show activate/activate="*

    Model  Routine  Status
    I1  BRTN    EXECUTE
    I1  GRTN    EXECUTE
Network idle, Simulation Clock= |5 nS
BEGIN-END

Groups a collection of commands to act as one command.

**FORMAT**

```
[label:] BEGIN  cmd1 [: ; ]
       cmd2 [: ; ]
       cmdn [: ; ]
    END
```

Default Qualifiers

None.

Optional Qualifiers

None.

**restrictions**

- You may not abbreviate the BEGIN and END keywords.

**DESCRIPTION**

The BEGIN-END block allows one or more commands to be treated as one command. BEGIN-END blocks are generally used in conjunction with the FOR, WHILE, UNTIL, IF-THEN, SELECT, and WATCH commands, but can also be used like any other interactive command at the SIM> prompt.

The commands enclosed in the BEGIN-END block are executed sequentially.

A BEGIN-END block is treated as one command entity by the ENABLE INDIRECT/NEXT=n command and by the CONTINUE command macro.

Whether the BEGIN-END block is entered at the SIM> prompt or is used in conjunction with another command, such as FOR or WATCH, DECSIM makes no checks on qualifiers, access lists, expressions, etc. until the END statement is entered. After the END keyword, DECSIM verifies each instruction and then executes them. Complete checking occurs every time the statement is executed.

You can attach a label to each block—for example, “Loop:BEGIN”.

Each command enclosed by the BEGIN-END block must follow line continuation rules (such as, two commands appearing on one line must be separated by semicolons).

At the syntax level, BEGIN-END pairs must be matched. It is useful to adopt an indentation formatting scheme to aid readability. Indenting text between BEGIN and END will also increase readability.

For more information see: Continuations (in Section 4.1.1).
EXAMPLE

WATCH A,B,C DO  
BEGIN  
PRINT A, B, C  
IF (A AND B) EQL 1 THEN HALT  
END

Control Flow Tracing

The following tracing is shown when the "DEBUG /LOOP" command is in effect.

SIN> FOR I FROM 1 TO 10 DO  
MORE> BEGIN  
MORE> IF I EQL 3 THEN LEAVE ALL  
MORE> PRINT I,'**2 = ',(I*I)  
MORE> END  
[ FOR-loop: I = 1 ]  
[ Entering block level 1 ]  
[ IF_TEST_COND is FALSE ]  
[ No ELSE alternative specified ]  
1**2 = 1  
[ Leaving block level 1 ]  
[ FOR-loop: I = 2 ]  
[ Entering block level 1 ]  
[ IF_TEST_COND is FALSE ]  
[ No ELSE alternative specified ]  
2**2 = 4  
[ Leaving block level 1 ]  
[ FOR-loop: I = 3 ]  
[ Entering block level 1 ]  
[ IF_TEST_COND is TRUE ]  
[ LEAVE ALL ]  
[ Leaving block level 1 ]  
[ End of FOR-loop: I ]
BREAK

Allows you to enter an interactive debug mode to inspect and modify signals in the network during simulation. Use BREAK in conjunction with WATCH to specify at what point the simulation is to be suspended.

SET BREAK
Suspends any current simulation and enters the interactive debug mode.

FORMAT [ SET ] BREAK

Default Qualifiers
None.

Optional Qualifiers
None.

restrictions
• The BREAK command can only be executed from within a watch_action or at the CTRL/C prompt.

DESCRIPTION

When the WATCH is triggered and DECSIM encounters a BREAK command in the watch_action, it prompts you with SIMBREAK> to enter DECSIM commands. In this mode, you can enter any command except SIMULATE. When you exit from this mode via the CANCEL BREAK command, the watch_action is continued, after which simulation resumes at the point at which it was stopped.

BREAK is strictly intended for interactive debugging. In batch mode, BREAK causes an informational message to be printed and then automatically uses the CONTINUE command.

For more information see:

CANCEL BREAK ENABLE BREAK RESUME
DISABLE BREAK CTRL/C B STEP
SHOW BREAK HALT WATCH
EXAMPLES

SIM> WATCH /STATEMENT:e_mod
MORE> BEGIN
MORE> EXAMINE sig
MORE> BREAK
MORE> END
%I, [SIMTXT] Watchpoint set at /STATEMENT:E_MOD\9 (000DCE1B)
%I, [SIMTXT] WATCH/TRACE Name is WATCH1
SIM>
SIM> WATCH /CALL=rtna /RETURN=rtna
%I, [SIMTXT] WATCH/TRACE Name is WATCH2
SIM>
SIM> CALL rtna
WATCH/CALL at RTNA
SIG<0> = 0
BREAK from WATCH /WATCHNAME=WATCH1 /STATEMENT:E_MOD\9 (000DCE1B)
SIMBREAK>
...
...
SIMBREAK> CANCEL BREAK
WATCH/RETURN at RTNA
SIM>
CANCEL BREAK

Terminates the current interactive debug session and returns to the simulation at the point at which it was interrupted. CANCEL BREAK is equivalent to the RESUME command macro.

FORMAT

CANCEL BREAK [/qualifier] [step_count]

Default Qualifiers
None.

Optional Qualifiers
/STEP
/STEP/INSTRUCTION

restrictions

• The CANCEL BREAK command must be executed interactively.

ARGUMENTS

step_count

Specifies the number of behavior language source lines to be executed in the currently executing routine before DECSIM returns to interactive debug mode. The default is 1.

Note: Stepping outside the routine is not allowed. You are returned to debug mode when the end of the routine is reached, even if the step_count is not exhausted.

QUALIFIERS

/STEP

Specifies that the exit from interactive debug mode is only for the time it takes to execute the specified number of behavior language lines. After that time, you are returned to debug mode. CANCEL BREAK/STEP is equivalent to the STEP command macro.

Note: You can use the /STEP qualifier only when the behavior language statement specified by the WATCH/STATEMENT command is about to be executed. It is not possible to step forward from any other type of WATCH trigger, including WATCH/CALL.

/STEP/INSTRUCTION

Specifies that the single-stepping action is in VAX machine instruction units rather than behavior language source lines.

For more information see:

DISABLE BREAK SET BREAK RESUME
ENABLE BREAK SHOW BREAK STEP
EX`APL`ES

SIM> WATCH /STATEMENT=9 DO BREAK
%I, [SIMTXT] Watchpoint set at /STATEMENT=E_MOD\9 (000DCE1B)
%I, [SIMWIN] WATCH/TRACE Name is WATCH1
SIM>
SIM> CALL rtna
BREAK from WATCH /WATCHNAME=WATCH1 /STATEMENT=E_MOD\9 (000DCE1B)
SIMBREAK> CANCEL BREAK /STEP
  BREAK after stepping to line 10
SIMBREAK> )CANCEL BREAK /STEP 999
  BREAK after stepping to end of routine at /STATEMENT=E_MOD\18 (000DCE9C)
SIMBREAK> CANCEL BREAK
SIM>
DISABLE/ENABLE BREAK

Temporarily disables or enables entry to interactive debug mode. When entry is disabled and a BREAK command is encountered in a watch_action, a message informs you that breaks are disabled; the remainder of the watch_action is executed as if you had entered a CANCEL BREAK command.

For more information see:
CANCEL BREAK  SET BREAK  SHOW BREAK

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>DISABLE BREAK</th>
<th>ENABLE BREAK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Qualifiers</td>
<td>None.</td>
<td></td>
</tr>
<tr>
<td>Optional Qualifiers</td>
<td>None.</td>
<td></td>
</tr>
</tbody>
</table>

restrictions
None.

EXAMPLES

SIM> WATCH /CALL=rtna DO BREAK
$I, [SIMWTN] WATCH/TRACE Name is WATCH1
SIM>
SIM> DISABLE BREAK
$I, [SIMTXT] Future SET BREAK commands will be ignored - until the next ENABLE BREAK command is issued.
SIM>
SIM> CALL rtna
    BREAK from WATCH /WATCHNAME=WATCH1 /CALL=rtna
    [ BREAKs are disabled ]
$I, [SIMTXT] continuing with remainder of WATCH action...
SIM>
SHOW BREAK

Displays the reason for entering debug mode and whether future breaks are enabled or disabled.

For more information see:
CANCELS BREAK ENABLE BREAK RESUME
DISABLE BREAK SET BREAK

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>SHOW BREAK  [ level_count ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Qualifiers</td>
<td>None.</td>
</tr>
<tr>
<td>Optional Qualifiers</td>
<td>None.</td>
</tr>
<tr>
<td>restrictions</td>
<td>None.</td>
</tr>
</tbody>
</table>

ARGUMENTS  
level_count  
Specifies the number of levels of breakpoints to show, starting with the most recent. The default is to show all levels.

EXAMPLES

SIM> WATCH/CALL=rtn DO BREAK  %I, [SIMWIN] WATCH/TRACE Name is WATCH1  SIM> CALL rtn  BREAK from WATCH /WATCHNAME=WATCH1 /CALL=RTNA  SIMBREAK>

...  
SIMBREAK> SHOW BREAK  1 BREAK from WATCH /WATCHNAME=WATCH1 /CALL=RTNA  [ BREAKs are enabled ]  SIMBREAK> )WATCH /RETURN=rtnb DO BREAK  %I, [SIMWIN] WATCH/TRACE Name is WATCH2  SIMBREAK> CALL rtnb  BREAK from WATCH /WATCHNAME=WATCH2 /RETURN=RTNB  SIMBREAK> SHOW BREAK  2 BREAK from WATCH /WATCHNAME=WATCH2 /RETURN=RTNB  1) BREAK from WATCH /WATCHNAME=WATCH1 /CALL=RTNA  [ BREAKs are enabled ]  SIMBREAK2> )RESUME  SIMBREAK> RESUME  SIM>

5-12
CALL

Interactively executes a routine in a behavior model.

**FORMAT**

```call routine_name [(parameter_list)]
```

**SHOW CALL**

Default Qualifiers
None.

Optional Qualifiers
None.

**restrictions**

- The CALL command is used with behavior routines only.

**ARGUMENTS**

*routine_name*
Name of a routine that may be prefixed by a series of labels identifying its location in the network.

*parameter_list*
Sequence of expressions, separated by commas. The number of expressions must match the number of formal parameters defined in a routine. The value of each expression is placed in the corresponding formal parameter.

**DESCRIPTION**

The CALL command initiates or continues the execution of behavior routines.

You can call a specific behavior model routine by specifying its *routine_name*.

You can continue an interrupted routine (perhaps caused by a WATCH command) by using the CALL command without specifying any *routine_name*.

The number of actual parameters must match the number of formal parameters. If not, an error message is issued by the DECSIM command language. Each parameter is a command expression. The actual parameter expression values are type-converted and extended, or truncated so that they match the width corresponding to the formal parameter type. The following table shows the conversion and the extension.

<table>
<thead>
<tr>
<th>FORMAL</th>
<th>Conversions</th>
<th>Extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE</td>
<td>0 1 2 3 U ?</td>
<td>-----------</td>
</tr>
<tr>
<td>2-state</td>
<td>0 1 0 0 0</td>
<td>ZXT</td>
</tr>
</tbody>
</table>
If a routine has a routine value returned, it is loaded into the system variable, %ROUTINEVALUE, which is accessible in any command language expression. However, this value is not automatically printed on the terminal upon completion of the routine. The return values are not stacked, so that only the last routine's value is accessible. Note that a routine that has no return value does not update %ROUTINEVALUE with an indeterminate value. If several nested levels of routines are called implicitly from the model code itself, the return values of the intermediate nested routines are not accessible using %ROUTINEVALUE. But, these values are accessible with the EXAMINE and/or the EVALUATE commands at the point of a WATCH/RETURN condition.

Timing is disabled when routines are executed from a CALL command. The interpretation of statements with timing is shown below. Assignments to global states are performed and are side effects of the routine call.

<table>
<thead>
<tr>
<th>STATEMENT</th>
<th>INTERPRETATION</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT expression;</td>
<td>no operation</td>
<td>routine execution continues</td>
</tr>
<tr>
<td>state=(delay) expression; state = expression;</td>
<td>assignments done immediately</td>
<td></td>
</tr>
</tbody>
</table>

The SHOW CALL command lists the routines that have been called but are pending completion. They are listed in the order in which they will be resumed.

For more information see:

- ACTIVATE
- EVALUATE
- EXAMINE

**EXAMPLES**

```plaintext
SIM> CALL start
SIM> CALL alu_eval(op1, op2)
SIM> EVALUATE %ROUTINEVALUE

Get return value from alu_eval routine
```

```plaintext
SIM> SHOW CALL
     Model  Routine  Status
    I1       ALU_EVAL  EXECUTE
```
COMBEH (Command Macro)

Allows you to compile a behavior model definition file. Qualifiers are placed after the filename and can be abbreviated to the first three letters. COMBEH is the same as COMPILE/BEHAVIOR.

FORMAT

COMBEH  filename [/qualifier ... ]

Default Qualifiers
/ERRORLIMIT=200
/LOAD
/NLB=filename.NLB
/NODEB
/NODIAI
/NOLIS
/NOLST

Optional Qualifiers
/DEB
/ERRORLIMIT=integer
/LIS
/NLB=filename
/PER
/SLB=filename

Development and Maintenance Qualifiers
/BLI
/COM
/DIAI
/EDT
/LST

restrictions

• The COMBEH command is used with behavioral models only.

QUALIFIERS

/DEB
/NODEB

Specifies whether to generate debugging information so that the source line debugging and array index checking facilities are available during simulation. The default is /NODEB.

/ERRORLIMIT=integer

Stops compilation after integer errors or warnings. You can use the keyword INFINITY for integer. Default limit is 200 errors. Warnings count as errors.
COMBEH (Command Macro)

/LIS
/NOLIS
Specifies whether to generate a list file from SX containing the source and
the symbol table. This file is called filename.LIS. The default is /NOLIST.

/NLB = filename
Declares an alternate file name for the network library file. The default
extension is .NLB.

/PER
Prints the performance measurements for SX, BLISS, and OBJMRRG.

/SLB=filename
Declares an alternate filename for the separate-compilation library file.

/SX='string'
Passes the string verbatim to COMBEH. For example:

SIMP> COMBEH file_name /SX='/EDT'

DEVELOPMENT
AND
MAINTENANCE
QUALIFIERS
These qualifiers are entered in quotation marks as values to the /SX= qualifier. For example:

SIMP> COMBEH file_name/SX='/BLI'

/BLI
Starts compilation with the BLISS (.BLI) file.

/COM
Resumes the command procedure starting with BLISS rather than SX.
The input file should be the file created by SX. This qualifier is used
mainly by developers.

/DIAG (option [,....])
/NODIAG
Specifies whether to generate an SX internal debugging file called
filename.DEB and writes selected internal data structure dumps to
it during compilation. The SX DIAG options are PARSE, SYNTAX,
IDENTIFIERS, SEMANTICS, DECLARATIONS, BIND, FOLD,
SUBSCRIPT, WIDTH, COERCCE, CALL, TEMPORARY, ALLOCATION.
The default is /NODIAG. See SXFETCH:SNM.TXT in “diag_switch” lines.
This qualifier is intended for developers only.

/EDT
Terminates the command procedure after SX generates the .BLI file. Saves
the .BLI file.

/LST
/NOLST
Specifies whether to generate a list file from BLISS containing the source
and the object code. This file is also called filename.LIS. The default is
/NOLST.
COMBEH (Command Macro)

DESCRIPTION
The COMBEH command macro creates a sub-process in which it calls the COMBEH.COM file from the DECSIM area on your system and compiles the specified files. The compilation procedure follows:

1. COMBEH runs SX, BLISS, and OBJMRG to compile behavior model definition files. It takes .BDS files and produces .NLB files.

2. These .NLB files must then be instantiated in a network instance file (.NET file).

3. The .NET file must then be compiled using the DECSIM COMPILE command to create a loadable network (the .NOB and .EXE files).

Note: If a word subscript is a literal or constant, COMBEH indicates when it is not in the declared range.

For more information see:

COMPILE  USE

EXAMPLES

SIM> COMBEH 2_bit_adder.bds

SIM> COMBEH 2_bit_adder /LIBRARY=adder.txt

COMBEH Macro Definition

SET MACRO /COMMAND /ABBREV=4 COMBEH [ $TEXT ] =
COMPILE/BEHAVIOR $TEXT
$ENDMACRO
COMPIL}

DESCRIPTION
Invoker the network processor (NETPRO) to compile the network description. COMPIL/BEHAVIOR can be used to compile behavior model definition files. Qualifiers can precede or follow the filename.

FORMAT
COMPIL [ Qualifier... ] filename [ Qualifier... ]

Default Qualifiers
/ERRORLIMIT=200
/LABEL_PIN
/LOAD

Optional Qualifiers
/BEHAVIOR
/DEBUG
/DEFAULT_LIBRARY=file
/ERROR_LIMIT=Integer
/EXECUTABLE=filename
/LIBRARY [=filename]
/LISTING [=filename]
/NETNAME=filename
/NETPRO='string'
/NOCOMPILE
/NOERROR_LIMIT
/NOLABEL_PIN
/NOLOAD
/PERFORMANCE
/PROCESS=filename
/QUIET
/STATISTICS [=type]
/SX='string'

ZYCAD-specific Qualifiers
/INPUT-PARTITION=file specification network_source_file
/OUTPUT_PARTITION=file specification network_source_file

restrictions
None.

ARGUMENTS
filename
The name of the .NET file.

QUALIFIERS
/BEHAVIOR
COMPIL/BEHAVIOR is used to compile a behavior model definition file into a library file with the extension .NLB. It is the same as COMBEH. All
COMBEH qualifiers can also be used with COMPIL/BEHAVIOR. See the
description of COMBEH in this chapter for more information.

/DEBUG
Creates a network with statement debugging information.

Behavior models must first be compiled with COMBEH filename/DEB
(or with COMPIL/BEHAVIOR/DEBUG filename) then with
COMPIL/DEBUG.

/DEFAULT_LIBRARY=file
Supplies an automatic network description language LIBRARY statement
before beginning processing of the network compilation. This is useful for
adding libraries that are not included in your network to the compilation.
The indicated file name must be an existing compiled–network–library
file (.NLB type). See also the CONFIGURE/LIBRARY command. This
qualifier may be used only once per COMPIL command.

/ERROR_LIMIT=integer
/NOERROR_LIMIT
/ERROR_LIMIT=integer stops compilation after the specified number
of errors or warnings. The default limit is 200 errors. You can use the
keyword INFINITY for integer. DECISIM counts warnings as errors.

/NOERROR_LIMIT is the same as /ERROR_LIMIT=INFINITY

/EXECUTABLE=filename
The file name on a /EXECUTABLE qualifier allows an alternate file name
to be specified for the .EXE file produced. An .EXE file is produced only
if there are behavior models in the network being compiled. Without this
qualifier, the file name is the same file name as the .NET file.

/INPUT_PARTITION=file_specification network_
source_file
A ZYCAD–specific qualifier pointing to a file containing instructions on
how to segment, or partition, the model network. This allows you to
specify what parts of your model are simulated in which ZYCAD S–module,
and can speed up simulation of large networks. See Appendix E for more
information on the input partition file.

/LABEL_PIN
/NO_LABEL_PIN
If /NO_LABEL_PIN is specified, the compiler does not use "label.pin"
format signal names. This speeds up NETPRO 10% – 15% when compiling
low–level models. (See Section 3.4.5.3 for more information on label.pin
names).

/LABEL_PIN is the default.

/LIBRARY [ = filename ]
The /LIBRARY qualifier must be supplied when you are compiling a .NET
file that is written in library format (such as GATEs.NET). Compiling a
library format .NET file produces a compiled–network–library file (default
extension .NLB).
Omitting the /LIBRARY qualifier when compiling a library format .NET file is not allowed. Similarly, a /LIBRARY qualifier is invalid if you supply it when compiling a .NET file that is a network format file.

The optional file name on a /LIBRARY qualifier allows an alternate file name to be specified for the .NLB file produced. The default file name is the same file name as the .NET file.

/Listing [ = filename ]
Produces a file of NETPRO error messages. The default extension is .LIS. This file does not echo the source file text as the COMBEH /LIS qualifier does.

/Load
/NoLoad
/LOAD creates and loads the .NOB file. This is the default qualifier.
/NOLOAD inhibits loading of the .NOB file.

/NetName = filename
The file name on a /NetName qualifier allows an alternate file name to be specified for the .NOB file produced. Without this qualifier, the file name is the same file name as the .NET file.

/NetPro = 'string'
Gives the NETPRO compiler the extra qualifiers verbatim from the string at the end of its command line. This qualifier is used mainly by maintainers.

/NoCompile
Restricts compilation to those files for which no .NOB file exists, and executes the USE command for existing .NOB files.

/Output_Partition = file_specification network_ source_file
A ZYCAD-specific qualifier that allows you to control the file specification of the partition output file (PAO). If this qualifier is not specified, the PAO is generated using your current directory, the file name of the partition input (PAI), and the extension PAO. If this qualifier is not specified and there is no PAI file, the PAO is generated using your current directory, the file name of your network, and the extension PAO. For information on the output partition file, see Appendix F.

/Performance
Produces a dump of NETPRO performance data in the .LIS file. This qualifier opens a .LIS file whether or not the /Listing qualifier was specified. This qualifier is used mainly by developers.

/Process = filename
Specifies the DECSIM process parameter file for DECSIM MOS simulations. This qualifier must be specified only when compiling a top level network that contains MOS to generate a .NOB file. It cannot be used to generate a .NLB file.
The process files are customized for each process. See SYS$DECSIM:DCSZMOS.PRC for an example file. This qualifier is required when compiling the MOS instance files (creating the MOS .NOB files).

/QUIET=(keyword)
Shuts off informational messages displayed during compilation. "Keyword" can be ALL or SDF (Structural Defaulting). /QUIET=(ALL) shuts off all informational messages. This is the default. /QUIET=(SDF) shuts off informational messages if a subblock name is not specified.

/STATISTICS [= type]
Displays additional compilation statistics on your terminal about the specified type. Normally, DECSIM sends the information to the .LIS file, provided you have used the /LISTING qualifier. There are two types: COUNTS and SIZES. COUNTS displays element counts by each element type. SIZES summarizes the sizes, in fullwords, of different data structures loaded in DECSIM. The default is both COUNTS and SIZES.

/SX = 'string'
Passes the string verbatim to COMBEH. For example:

SIM> COMPIL/BEH/SX=`/BLI` behavior_file_name

This option is intended for maintainers.

**DESCRIPTION**

The COMPIL command has four principal functions:

- Processes a network description so that it can be simulated.
- Produce compilation listings and cross-reference listings of the network.
- Creates a library of models for future use.
- Loads .NOB files for simulation.

The COMPIL command causes DECSIM to run the network processor program, NETPRO, which compiles the files named by the user. If no errors are detected during compilation and if you have selected the appropriate qualifier, DECSIM loads the newly compiled network into the active memory image, ready for simulation.

**Note:**

- Networks containing warnings are loaded, but networks containing errors are not loaded.
- Any previously loaded network and its states are lost when the COMPIL command is given. However, MACRO and STATE definitions are retained; STATISTICS is reset.
- Use CTRL/C, not CTRL/Y, to abort a NETPRO compilation.
**ZYCAD Mode**

To prepare your network in ZYCAD mode you must compile the network with the primitive model library ZGATES.NLB, rather than with GATES.NLB. The ZGATES.NLB is released with DECSIM, and imitates GATES.NET for normal logic gates. The only models containing equation statements are those in ZGATES.NET. All other models must be synthesized from this collection. The rules for constructing models are:

- Only add primitive ZYCAD Gate Model instantiations.
- If you must extend ZGATES.NET, contact the DECSIM/ZYCAD team.

For more information see:
- COMBEH
- USE
- ZYCAD
CONFIGURE

Allows you to display or set up defaults for DECSIM system files.

FORMAT

[SET] CONFIGURE /qualifier [...] SHOW CONFIGURE

Default Qualifiers
None.

Optional Qualifiers
/HELP=logical
/LIBRARY=filename
/NETPRO=filename
/NOLIBRARY

restrictions

- You must use at least one qualifier with the [SET] CONFIGURE command.

QUALIFIERS

At least one qualifier must be used. In each case, the file name (log name) is not checked until it is executed.

/HELP= logical
Determines the directory that contains DECSIM help files.

/LIBRARY= filename
/NOLIBRARY
Stores default library file specifications for all following COMPILe commands. More than one CONFIGURE/LIBRARY may be given; in this case, a list of files is given to NETPRO. 
/NOLIBRARY removes all entries from the default library file list.

/NETPRO= filename
Determines the file that contains the NETPRO compiler.

DESCRIPTION

System developers access software under development with the CONFIGURE command. For example, to facilitate a site field test, the CONFIGURE command could be used in DECSIM.SYS (for a complete site), DECSIM.INI (for a group or single user), or interactively to allow certain users to access the field test version of the software.

You are not required to define the logical names before you invoke DECSIM; you can use the DECSIM DCL command. If the logical names are not specified, the defaults are those shown in the example at the end of this section.
With CONFIGURE/LIBRARY, you set up the default NETPRO libraries for COMPILE commands. The last library entered is the first library searched. SHOW CONFIGURE displays the current programs and files that comprise the DECSIM system.

For more information see:

COMPILE SHARE
HELP SPR

EXAMPLES

SIM> SHOW CONFIGURE
DECSIM Version V5.4-3806
SIM$HELP: DECSIM$SYS:DECSIM.HLB
NETPRO
SIM$NETPRO: DECSIM$SYS:NETPRO
Default LIBRARY files (list is searched from the bottom up):
DECSIM$SYS:GATES
SX
SIM$SX: DECSIM$SYS: SX.EXE
SIM>
SIM> SHOW CONFIGURE
DECSIM Version V5.4-3806
SIM$HELP: DECSIM$SYS:DECSIM.HLB
NETPRO
SIM$NETPRO: DECSIM$SYS:NETPRO
Default LIBRARY files (list is searched from the bottom up):
USER$LIB:GATES
USER$LIB:ZMOS
SX
SIM$SX: DECSIM$SYS: SX.EXE
CONTEXT

Specifies the default machine descriptor for fault simulation.

FORMAT

[SET] CONTEXT  (machine descriptor)
SHOW CONTEXT
CANCEL CONTEXT

Default Qualifiers
None.

Optional Qualifiers
None.

restrictions

- The CONTEXT command is used only for fault simulation.
- This command changes the default machine descriptor for the EXAMINE and DEPOSIT commands only.

ARGUMENTS

machine descriptor
A single machine descriptor enclosed in brackets.

DESCRIPTION

During fault simulation, the SET CONTEXT command defines the default network through which you access signals. Normally, the default context is the good machine or [PRIMARY]. SET CONTEXT allows you to specify a faulty machine (one in which you have inserted faults) as the default network. You may use SET CONTEXT seven consecutive times without using CANCEL CONTEXT.

Faults dropped during simulation can cause a context set with the CONTEXT command to become invalid. In this case, the current context is canceled and the next context on the stack is reevaluated. This process is repeated until the context is valid or you are back at the PRIMARY machine.

CANCEL CONTEXT returns the context to the one set by the previous CONTEXT command or to [PRIMARY]. If the previous CONTEXT is no longer valid, it is also canceled.

SHOW CONTEXT displays the current default machine descriptor.

For more information see:

DEPOSIT  FAULT
EXAMINE  machine descriptors
EXAMPLES

SIM> ! The network is composed of two independent buffers. The
SIM> ! following test inserts a fault on the output of one buffer. A
SIM> ! deposit is then made to the faulty (concurrent) machine. Note
SIM> ! that the value associated with the faulty machine is now
SIM> ! independent of the value associated with the original fault.
SIM> ! Note also that the effect of the deposit is not seen in the
SIM> ! examine.
SIM> ! IN1_E1 o--------|>----------------- o OUT_E1
SIM> !
SIM> ! IN1_E2 o--------|>--------(SA1)------ o OUT_E2
SIM> !
SIM> !
SIM> USE depdeb:tsdep5

Reading File: DSUS:[TEST]TSDEP5.NOB;2
%I-TRS, Time Resolution = 1 nS, default unit is nS
SIM> SHOW CONTEXT
Context = {PRIMARY}
SIM> SYMBOL
IN1_E1 = U
IN1_E2 = U
OUT_E1 = U
OUT_E2 = U
SIM> FAULT/OUTPUT/STUCK:(1) out_e2
SIM> DEPOSIT in1_e2 = 1
SIM> DEPOSIT in1_e1 = 1
SIM> SET CONTEXT [sa1:out_e2]
Context = {SA1:OUT_E2}
SIM> DEPOSIT in1_e2 = 0
SIM> SIM 10
Network idle, Simulation Clock: |10 nS
SIM> SYMBOL
IN1_E1 = 1
IN1_E2 = 0
OUT_E1 = 1
OUT_E2(OSA1:OUT_E2) = 1
SIM> SHOW CONTEXT
Context = {SA1:OUT_E2}
SIM> CANCEL CONTEXT
! Pop the CONTEXT stack
Context = {PRIMAR Y}
SIM> SYMBOL
IN1_E1 = 1
IN1_E2 = 0
OUT_E1 = 1
OUT_E2 = 0
CONTINUE (Command Macro)

Resumes reading and execution of a suspended indirect file.

**FORMAT**

CONTINUE

CONTINUE [/qualifier]

**Default Qualifiers**

None.

**Optional Qualifiers**

/NEXT [=integer]

**restrictions**

- This command can be executed only after an error, or after a DISABLE INDIRECT or PAUSE command.

**QUALIFIERS**

/NEXT [=integer]

Pauses again after executing one or more commands from the indirect file. If several commands are placed on one line, only one is executed. BEGIN-END blocks are treated as one command. If an argument is given, DECSIM pauses after executing integer commands.

**DESCRIPTION**

This command continues executing the current indirect file after a DISABLE INDIRECT, a PAUSE, or an error.

If a SIMULATE command was in progress when the interruption occurred, the CONTINUE command allows the SIMULATE command to finish execution before returning to the indirect file. To complete the SIMULATE command without returning to the indirect file, use the form:

   CONTINUE/NEXT=0

For more information see:

CTRL/C C @, SET INDIRECT PAUSE
MACRO SIM> (prompt)
CONTINUE (Command Macro)

EXAMPLES

SIM> @/echo test1.log
SIM> use network1
Reading File: DSUS:[FILLMORE]NETWORK1.NOB;5
Time Resolution = 100 PS, default unit is ns
SIM> trace/odo sig1, sig2, sig3
%I, [SIMWTN] WATCH/TRACE Name is TRACE1
SIM> pause
[ PAUSE at simulation time 0 ns ]
[ "ENABLE INDIRECT" to proceed with file DSUS:[FILLMORE]TEST1.LOG;3
SIM> continue
SIM> dep sig1, sig2 = 0
SIM> simulate 20
Simulation Clock: |20 ns
!End of indirect file DSUS:[FILLMORE]TEST1.LOG;3
SIM>

CONTINUE Macro Definition

SET MACRO /COMMAND /ABBREV=4 CONTINUE =
ENABLE INDIRECT
SENDMACRO
DCL

Implements a few simple VAX/VMS DCL commands within DECSIM.

**FORMAT**

DCL DEASSIGN  logname
DCL DEFINE   logname equivalence_name
DCL SET DEFAULT directory_name
DCL SHOW DEFAULT
DCL SHOW LOGICAL   logname

**Default Qualifiers**

*None.*

**Optional Qualifiers**

*None.*

**restrictions**

- The keywords that work with this command are limited to those listed above.

**DESCRIPTION**

The DCL command is used with a limited number of keywords to implement the VAX/VMS DCL counterparts within DECSIM.

DCL DEASSIGN logname cancels a logical name assignment made with the VAX/VMS ALLOCATE, ASSIGN, or DEFINE commands or the DECSIM DCL DEFINE command.

DCL DEFINE logname equivalence_name is used to equate character strings with file specifications or logical names.

DCL SET DEFAULT directory_name changes the default device and/or directory name for the current process. The new default is applied to all subsequent file specifications that do not explicitly give a device or directory name.

DCL SHOW DEFAULT displays the current default device and directory name.

DCL SHOW LOGICAL logname displays the current equivalence name assigned to the specified logical name.
DEBUG

Affects and shows certain system and informative features in the DECSIM command language and simulator, principally for developers.

FORMAT

[SET] DEBUG [/qualifier] [access_list]
SHOW DEBUG
CANCEL DEBUG access_list

Default Qualifiers
/NOEXPAND
/NOLOOP

Optional Qualifiers
/LOGNAME=logfile [...]
/MOS
/MOS=(keyword [...])
/EXPAND
/LOOP

restrictions
- The DEBUG command is used primarily by developers and is mostly undocumented.

ARGUMENTS

access_list
A list of signals that the DEBUG command traces. For consecutive DEBUG commands, the access list of the DEBUG command is appended to the access list of the previous DEBUG command, unless you remove signals from the access list with CANCEL DEBUG access_list.

QUALIFIERS

/EXPAND
/NOEXPAND
Enables a detailed trace of macro expansions. The following macro expansion characteristics are traced:

macro expansion (start, text, end)
formals
directed:

/LOGNAME=logfile
/LOGNAME=(logfile,...)
Directs simulation debug information to a log file. If a logfile has previously been set up with a LOG command, output is routed to this log file. Otherwise, an implicit SET LOG command is performed, which opens a log file. Multiple lognames can be used to print to multiple files by enclosing the list in parentheses. The log file remains open until it is explicitly closed with CANCEL LOG.
/LOOP
/NOLOOP

Enables a detailed trace on the following control constructs:

- IF THEN ELSE
- LEAVE (leave a block)
- for LOOP
- BLOCK LEVEL (ENTERING AND LEAVING)
- while/until
- halt
- select
- watch
- ASSIGNMENT
- BEHAVIOR ROUTINE CONTROL FLOW (CALL, RETURN, WAIT, RESUME)

The following qualifier and keywords apply to MOS simulations only:

/MOS

Specifies that no MOS characteristics are to be displayed.

/MOS=(keyword [ , ... ])

Specifies that keyword information be displayed for all MOS nodes or, if an access list is supplied, for those nodes in the access list. If the lists are empty, all MOS nodes are printed. Multiple DEBUG commands disables enabled keywords.

Keywords include:

- CHARGE_SHARE
  Displays the results of the capacitance equation.

- DELAY
  Displays conductance and capacitance for delay calculation.

- RECHECK
  Displays a more detailed calculation of the state of a node with enabled charge sharing. The node must be driven by both active transistors and by charge sharing.

- STATE
  Displays the results of the resistance calculation.

See also:

- BREAK
- RESUME
- WATCH/STATEMENT
- CANCEL
- WATCH/CALL
- WATCH/RETURN
- BREAK

DESCRIPTION

You use the SET DEBUG command to specify the qualifiers for printing internal simulator states.

Note: The DEBUG command is a developer tool with many undocumented features. For more detailed information, send mail to DECSIM::DECSIM_SUPPORT.
DEPOSIT

Allows you to input values to specified one bit signals. Enables you to force both primary inputs and internal nodes of scalar signals to a new state. The abbreviation for DEPOSIT is D.

FORMAT

DEPOSIT  [/qualifier] access_list [= wave_element ] [access_list = wave_element ...]
DEPOSIT/FREE  access_list
DEPOSIT/PIN  access_list=state

Default Qualifiers
/NOFINAL_RELEASE

Optional Qualifiers
/FINAL_RELEASE=argument
/FORCE
/FREE
/PIN

restrictions

• You cannot DEPOSIT a value to a vector.
• Further restrictions apply in ZYCAD mode. See DESCRIPTION below.

ARGUMENTS

access_list
Signal name, or a list of signal names separated by commas. These signals must be one bit wide. Signals specified with a wildcard should be enclosed in double quotation marks.

wave_element
Wave_element is one of the following:

state
[state]: time
repeating_wave
machine descriptor

where repeating_wave is:

[ wave_element [ , wave_element ... ] [count] ]

count
An integer or the keyword INFINITY. Default is one (1).

time
The time interval to delay before the next state is set. Absolute time (the simulator clock reading) may be indicated with a time. Absolute time
is allowed only for nonrepeating portions of the waveform (for example, count=1).

**state**
A signal state (0 or L, 1 or H, Z, U, and ? (free)).

**Note:** DECSIM interprets the signal states L, H, Z, and U as signal names unless you follow them with #2. For example, U represents the signal or variable of that name. U#2 represents the state "undefined". The question mark character must also be followed with #2, so that DECSIM interprets it as meaning "free."

If state is omitted, the complement of the previous state is assumed. The complement of an undefined (U) is a high impedance (Z) and the complement of a Z is a U. No error is generated.

State can be an expression containing other variables and operators. All state expressions are evaluated immediately, and the result is used regardless of when the transition takes place.

---

**QUALIFIERS**

/FINAL_RELEASE=argument

/NOFINAL_RELEASE

Used with /FORCE to specify what happens to a signal at the end of a waveform. The argument can be one of the following:

- **NONE** – specifies that the signal remain forced until you use the DEPOSIT/FREE command or drive the signal with a DEPOSIT or PATTERN command. NONE is the default.
- **ACTIVITY** – specifies that the signal keep its forced value until gate activity changes the signal value. For MOS, gate activity refers only to pull-up and pull-down stacks directly connected to the node.
- **EQUALITY** – specifies that the signal remain forced until gate activity makes its value identical to the forced value. For MOS, gate activity refers only to pull-up and pull-down stacks directly connected to the node.
- **time** – specifies that the signal be free a specific amount of time after the waveform ends.

/NOFINAL_RELEASE is equivalent to /FINAL_RELEASE=NONE. This is the default.

/FORCE

Forces a signal to have a specified value, thus overriding the real value of that signal. /FORCE is the only way to deposit a value to an internal node.

In fault simulation, /FORCE applies to both the good machine and the faulty machines.

/FREE

Removes the effect of /FORCE or /PIN. At the gate level, the freed signal changes to the state it would have been without the /FORCE with 0 delay. At the MOS level, the state of the signal is reevaluated and events are scheduled with actual delays.

5–33
/PIN
Deposits the specified state to the specified input pin. Simulation then carries that value through a device without forcing the entire fanout to that state. This is different from DEPOSIT/FORCE, which finds the driving output and deposits the state there, and which affects all of the specified signal's fanout.

Since DEPOSIT/PIN can be applied to a model's input, and that model may be at some level higher than the bottom of the hierarchy, the value will be applied to all input pins within that model that are tied to the named pin.

You cannot use a repeating waveform with /PIN. This qualifier may not be used in conjunction with /FORCE or with /FREE. To free the pin, use DEPOSIT/PIN signal_name=?#2.

DESCRIPTION
The DEPOSIT command permits you to set and change the state of signal nets from the command mode.

All elements of the access list are set equal to the initial state of the waveform.

Example
DEPOSIT CLOCK H=[0, [1:3, 0:2] INF]

In the example, Signal CLOCK H is set up to repeat a sequence forever. The signal is assigned a starting value of 0, waits 3ns before going high, then is high for 2ns before going low again. 1 and 0 could be omitted from the example without affecting the results.

An assignment of one value to several signals is allowed.

Example
DEPOSIT A, B, C=1

If you want to DEPOSIT a value at a later time depending on the transition of another signal, use the following command:

WATCH /ONCE signal DO DEPOSIT x=value;

You can use the question mark followed by pound sign 2 (?#2) to mean, "Free the signal if it's forced; otherwise, give it a value of U (undefined)."

In the following example, S is assigned a value of 1 for 25 nanoseconds, then is freed for 75 nanoseconds, for an infinite time.

Example
DEPOSIT/FORCE S = [1, [#2:25, 1:75] INF]

The following example complements the value of the freed signal, and then forces it to that value.

Example
DEPOSIT/FORCE S = [#2, :5]

Machine descriptors can also be used with the DEPOSIT command. See Section 4.3.8 for more information. The syntax is as follows:

SIM> DEPOSIT A1 (machine_descriptor) = state
For more information see:

CONTEXT EVALUATE PATTERN

ZYCAD Mode

In ZYCAD mode, the DEPOSIT command is converted to pattern stimuli at simulate time. The following restrictions apply:

- All deposits for a particular simulation run must be entered before using the SIMULATE command.
- Deposits remain in effect from one ZYCAD simulation run to another. You can add new deposits and overwrite old ones between simulation runs.
- DEPOSIT can drive a high impedance state (Z#2) only on primary inputs that drive buses.
- The complement of a U#2 is a U#2.
- The complement of the free character (?#2) is meaningless and generates an error at run time.
- When the INFINITE keyword is used in a DEPOSIT waveform, a time is required on the SIMULATE command.
- The only valid arguments for the /FINAL_RELEASE qualifier are NONE and time.

EXAMPLES

SIM> D RUN=0, LATTER=1:345 NS !sets run to 0 now, and latter to 1 at 345)
SIM> D CLK= [0:15 NS, 1:20NS] INF
SIM> D CLK_SQUARE = [:10] INF
SIM> D SIGNAL = [:5, :4, :7, [U#2:5 NS, 0:5]] 2
DETECT

Describes where in the network to place observation points, which are used to do one of the following:

- Determine fault coverage in fault simulation. (See /FAULT)
- Find oscillating nodes in good simulation. (See /OSCILLATION)
- Find which nodes change state in good simulation. (See /TOGGLE)

Observation points can be placed in MOS models, Structural models, and in memory, but not in Behavior models.

FORMAT

[SET] DETECT [/qualifier...] access_list

Default Qualifiers

/CONTINUOUS_VISIBILITY
/DETECTNAME = DETECTn
/DROP_CONDITION = FIRM
/FAULT
/FIRM_DETECTION = (0:1,1:0)
/LOGNAME = LOGNAMEn
/MIN_PULSE_WIDTH = 0
/NOCOMPRESSED
/NOMESSAGE_CONDITION
/NOPOSSIBLE_DETECTION
/REPORT = (SETUP:REFERENCE NOCONCURRENT)
          (WINDOW:REFERENCE NOCONCURRENT)
          (PULSE:REFERENCE NOCONCURRENT)
/SEVERITY = INFORMATIONAL
/STATIC
/WINDOW = none

Default Qualifiers (with /TOGGLE)

/COVERAGE = (0,1)
/THRESHOLD = 1

Required Qualifiers (with /OSCILLATION)

/STROBE = signalname or /EVERY = time
/THRESHOLD = 1

Optional Qualifiers

/BRIEF
/COVERAGE = (state1,state2,...)
/DETECTNAME = detect_name
/DROP_CONDITION = (good_signal :fault_signal,...)
/DYNAMIC
/ERRORLOG = logname
/EVERY = time
/FIRM_DETECTION = (good_signal :fault_signal,...)
/FROM = time
/FULL
/HALT
/INDEX
/LOGNAME=logname
/MESSAGE_CONDITION = (good_signal : fault_signal, ...)
/MINIMUM_PULSE_WIDTH = time
/NOCONTINUOUS_VISIBILITY
/NODROP_CONDITION
/NOFIRM_DETECTION
/NOLOG
/NON_DETECTION = (good_signal : fault_signal, ...)
/OSCILLATION
/POSSIBLE_DETECTION = (good_signal : fault_signal, ...)
/REPORT = (timing_violation_type,...)
/SCOPE = scope_name
/SETUP = time
/SEVERITY = message_severity_level
/TOGGLE
/WINDOW = time

ZYCAD-specific Qualifiers
/COMPRESSED

restrictions

- Observation points can be placed in MOS models, structural models, and in memory, but not in behavior models.
- Further restrictions apply in ZYCAD mode. See DESCRIPTION below.

ARGUMENTS

access_list
A list of outputs where fault detectors can be inserted.

QUALIFIERS

/BRIEF
Abbreviates the information that is sent to a log file or displayed on the screen during fault simulation. /BRIEF records when the fault was detected, the type of fault, the name of the faulty signal, and where the fault was detected. This qualifier decreases the size of the log file, which makes postprocessing of fault simulation log files faster. In ZYCAD mode, no strength or state information is reported.

/COMPRESSED
/NOCOMPRESSED
/COMPRESSED is a ZYCAD-specific qualifier that compresses the format of the .ZOF file so that there are no spaces between the traced signals, and there is only one header for each horizontal page. The .ZOF file displays the results of the specified signals for specified simulation times. Those signals that do not fit on the page are reported, just like the other signals, on a new page (horizontal page). To read the .ZOF file, place the "horizontal page" for each succeeding set of signals to the right of the previous one, to form a page wider than a printer can handle.

/NOCOMPRESSED is the default qualifier.
DECSIM accepts the format specified by the last DETECT command entered before the job is submitted to the Logic Evaluator as the report format.

/CONTINUOUS_VISIBILITY
NOCONTINUOUS_VISIBILITY
A fault that remains visible at a detection point during multiple strobe assertions will be reported as a fault each time the strobe is asserted until the fault is dropped.

/CONTINUOUS_VISIBILITY is the default.

/COVERAGE = (state,[state,...])
Specifies the states that the signals must change to in order for a detection to be made with /TOGGLE. The signals must change to all of the listed states to be detected. By default, /COVERAGE = (0,1). (See /TOGGLE.)

/DETECTNAME = detect_name
Assigns a unique name to the set of detectors that are specified with a DETECT command. This name can be specified with CANCEL DETECT detect_name, ENABLE DETECT detect_name, DISABLE DETECT detect_name, and SHOW DETECT detect_name.

By default, DECSIM assigns a name to detectors associated with each DETECT command. The default names take the form of DETECTn, where n represents an integer. Use SHOW DETECT to identify the detect_name associated with the current set of detectors.

/DROP_CONDITION = (good_signal :fault_signal, ...)
/NODROP_CONDITION
Specifies the conditions that cause faults to be dropped after they have been detected. These fault conditions are represented by a keyword or an argument pair list.

There are two keywords, FIRM and POSSIBLE, that you can use to specify the conditions for dropping a fault when you have used one or both of the qualifiers /POSSIBLE_DETECTION and /FIRM_DETECTION. The keyword FIRM indicates that the conditions for dropping faults are specified by the /FIRM_DETECTION qualifier, and the keyword POSSIBLE indicates that the conditions for dropping faults are specified by the /POSSIBLE_DETECTION qualifier.

The argument pair list specifies the signal state of the good machine and the signal state of the faulty machine for all drop conditions. For example: (1:0,Z:0) specifies that DECSIM should drop a detected fault when the good machine is 1 and the faulty machine is 0, and when the good machine is Z and the faulty machine is 0.

This qualifier should be used in conjunction with /FIRM_DETECTION and /POSSIBLE_DETECTION; however, DETECT/FIRM_DETECTION and DETECT/POSSIBLE_DETECTION do not override a drop condition specified in a previous DETECT command.

/DYNAMIC
Specifies that strobes are to be interpreted as levels rather than edges. With this qualifier, the detector becomes active after the rising edge of the strobe, and continues to be active as long as the strobe is high. Fault-free and faulty node transitions are recorded during the time that the strobe
is asserted. Fault detections (as defined with /POSSIBLE_DETECTION and /FIRM_DETECTION) are reported when the strobe goes to 0. This is in contrast to /STATIC, which does not activate the detector for a period of time, but only on the rising edge of the strobe.

/DYNAMIC must be used with /STROBE or /EVERY. If /EVERY is used, the duration of the asserted strobe must be specified with the /WINDOW qualifier. If /STROBE is used, /WINDOW is not allowed; the effective "window" is the time that the strobe signal remains asserted.

/ERRORLOGNAME = logname
/ERRORLOGNAME=(logname,...)
Names a log file for writing error messages generated by detect points during simulation. If a logname has been previously set up with a LOG command, output is routed to this log file. Otherwise, an implicit SET LOG command is executed, which opens a log file. Generally, the same log file that was specified to record the fault simulation is specified here.

/EVERY=time
Specifies the interval to be used in comparing the good machine to the faulty machine for those detection points named in the DETECT command access_list. /EVERY generates a periodic detection strobe starting with the current time or with the time specified with the /FROM qualifier. /EVERY is also used in good simulation with /OSCILLATION. (See /OSCILLATION.)

Note: DETECT/EVERY differs from WATCH/EVERY and TRACE/EVERY. With DETECT, the action starts at the time specified with the /FROM qualifier. With WATCH and TRACE, action starts with the current simulator clock time.

/FAULT
Indicates that DECSIM is to detect fault effects at the detection points.

/FAULT is the default. /OSCILLATE and /TOGGLE override /FAULT.

/FIRM_DETECTION = (good_signal :fault_signal, ...)
/NOFIRM_DETECTION
Specifies the conditions for which definite fault detections are reported. These conditions are represented by pairs of signals: the signal state of the good machine followed by a colon, and the signal state of the faulty machine. Lists of these pairs are separated by commas. For example: (0:1,1:0,Z:1).

See the DETECT command description for a list of the nine signal states that can be specified. See also /POSSIBLE_DETECTION and /NON_DETECTION.

/FROM=time
Specifies the starting time to be used in comparing the good machine to the faulty machine for those detection points named in the DETECT command access list. It is also used in good simulation with /OSCILLATION. (See /OSCILLATION.)
/FULL
Currently implemented in ZYCAD mode only. Reports full information to the .ZOF file about the detect points, including state and strength information.

/HALT
Stops simulation when DECSIM finds an oscillating signal with DETECT/OSCILLATION. Used in good simulation only.

/INDEX
Assigns a unique numeric identifier to each inserted detector so that the simulation output contains numbers instead of names. /INDEX shortens the time of simulation because no time is spent on name generation.

The log file DECSIM produces when /INDEX is specified with /LOG can be postprocessed by other tools.

For a list of fault indexes, see SHOW FAULT/INDEX. For a list of detector indexes, see SHOW DETECT/INDEX.

/LOGNAME=logname
/LOGNAME=(logname,...)

/NOLOG
Specifies the destination of the output. If a logname has previously been set up with a LOG command, output is routed to this log file. Otherwise, an implicit SET LOG command is performed, which opens a log file. The logname specification is a logical name. DECSIM gives DETECT log files an extension of .DET. This can be overridden by specifying a file extension.

Multiple lognames can be used to print to multiple files by enclosing the list in parentheses.

The logfile cannot be canceled with CANCEL LOG unless the DETECT has also been canceled.

/NOLOG inhibits the reporting of results. You can use SHOW FAULT/DETECTED and SHOW FAULT/UNDETECTED to see a list of detected and undetected faults.

In ZYCAD mode, /NOLOG produces an abbreviated .ZOF file containing the optional SHOW FAULT data and the standard run summary data.

/MESSAGE_CONDITION = (good_signal :fault_signal, 
...)

/NOMESSAGE_CONDITION
Describes signal–state conditions that will produce messages when one of the conditions occurs at a DETECT point. The condition is specified by a pair of signal states: the state of the signal in the good machine, followed by a colon (:), and the state of the signal in the faulty machine. Lists of these pairs are separated by commas. For example: (1:0,0:1,Z:U).

All messages will be output at the severity–level described by one of the keywords used as an argument to the /SEVERITY qualifier. (See the /SEVERITY qualifier description.)
/MIN_PULSE_WIDTH = time

Defines the minimum time that must elapse between multiple transitions within a window in order for the transition to be seen by a tester. This qualifier is used for signal states that involve transitions to or from T0, T1, TU, and TZ, which are recognized only when they are specified as arguments with /FIRM_DETECTION and /POSSIBLE_DETECTION. Since transitions to or from these states can be seen only when level-sensitive detection is enabled, the /DYNAMIC qualifier must be specified also. By default, /MIN_PULSE_WIDTH = 0.

If a pulse does not meet the minimum pulse width, DECSIM treats it as a window violation. (See /REPORT for more information on how these violations are reported.)

Since a tester is only capable of detecting pulses that exceed a certain minimum width, the specified time used with this qualifier is dependent on your tester's specifications. However, the time value specified by /MIN_PULSE_WIDTH must not exceed the time value specified with /WINDOW.

---

**EXAMPLE**

DETECT/MIN_PULSE_WIDTH=p/WINDOW=w/FIRM_DETECT = (0:T0,T0:0)

```
fault-free 1--+(<----w---->)  +++++|
signal 0                   |         |
        1 +--------|
faulty signal 0 (------p-----)  +--------|
              1 |
strobed signal 0 <|------w------|     +--------|
                1 |<------w------>
                DETECT NON-DETECT
```

DETECT/MIN_PULSE_WIDTH=p/WINDOW=w/FIRM_DETECT = (0:T0,T0:0)

```
fault-free 1--+(<----p---->)  +--------|
signal 0                   |
        1 +---------|
faulty signal 0(------p-----)  +--------|
              1 |
strobed signal 0 <|------w------|     +--------|
                1 |<------w------>
                DETECT NON-DETECT
```

---

**EXAMPLE**

DECSIM does not count a transition through the U state as a detection when the final transition within the window interval is not a U.

```
fault-free 1--+(<----w---->)
signal 0 |                  |
          1 |-------------------|
          1+-------|
faulty signal 0 (<-----w---->) | (------p-----)  +--------|
              1 |
strobed signal 0 <|------w------|     +--------|
                1 |<------w------>
                NON-DETECT NON-DETECT
```

5-41
/NON_DETECTION = (good_signal: fault_signal, ...)

Disables the detection of the specified pairs of signal states in the good and faulty machines.

For example, if (U:0) were specified, then there would be no detection of a fault if the good machine's detection point were undefined and the faulty machine's detection point were 0.

See the DETECT command description for a list of the nine signal states that can be specified. See, also, /FIRM_DETECTION and /POSSIBLE_DETECTION.

/OSCILLATION

Used for good simulation only. Checks the number of times a node goes through a transition in a specified period of time. If the number exceeds the acceptable number of transitions for the specified time period, DECSIM issues the message "Excessive events scheduled on the node: nodename."

The acceptable number of transitions is set by the qualifier /THRESHOLD = integer. The time period is specified with /FROM = time, /EVERY = time, and /STROBE = signalname. /FROM specifies a starting time for checking the observation points. /EVERY specifies a periodic check of the observation points. /STROBE indicates that the detection starts when the specified signal changes from 0, U, or Z to 1 and stops when the signal changes to a state other than 1.

/HALT, /LOGNAME = logname, /DETECTNAME = detect_name are additional qualifiers used with /OSCILLATE. See the qualifier descriptions for more information on their use.

/POSSIBLE_DETECTION = (good_signal: fault_signal, ...)

/NOPossible_DETECTION

Specifies the conditions for which POSSIBLE fault detections are reported. These conditions are represented by pairs of signals: the signal state of the good machine, followed by a colon and the signal state of the faulty machine. Lists of these pairs are separated by commas. For example: (0:1,1:0,Z:1).

Possible detections do not cause the percent coverage to rise.

See the DETECT command description for a list of the nine signal states that can be specified. See, also, /FIRM_DETECTION and /NON_DETECTION.

/REPORT = (timingViolation_type,...)

Controls reporting of timing related errors caused by the fault–free circuit or one or more faulty circuits.

There are three types of timing violations that can be specified with one of three keywords: WINDOW, SETUP, and PULSE. A window error is caused by any unexpected transition that occurs within the WINDOW time associated with a specific detection point. A setup error is any transition that occurs within the SETUP time associated with a specific detection point. A pulse error is any sequence of expected transitions that occur within the WINDOW time associated with a specific detection point and
that are not separated by at least the time specified by /MIN_PULSE_WIDTH.

The keywords WINDOW, SETUP, and PULSE take two arguments. The arguments are chosen from the following keywords: REFERENCE, NOREFERENCE, CONCURRENT, and NOCONCURRENT.

REFERENCE and NOREFERENCE specify whether DECSIM reports errors detected in the good machine. By default, DECSIM reports errors in the reference machine.

CONCURRENT and NOCONCURRENT specify whether DECSIM reports errors detected in the faulty machines. By default, DECSIM does not report these errors.

The two arguments are separated with a comma and placed within parentheses ( ).

EXAMPLE

DETECT/REPORT= (WINDOW= (NOREFERENCE, CONCURRENT),
SETUP= (NOREFERENCE, CONCURRENT),
PULSE= (NOREFERENCE, CONCURRENT))
/SETUP = s /WINDOW = w /MIN_PULSE_WIDTH = m
faulty signal good signal strobed signal
1------ +------- +----------+-------+-----| 0   I
1----------- +------------------|| 0   I
1<--s-->   +--------<--s-->+--------+<--w-->+|<--m-->|
setup|<--w--> |______|<--w-->|__|<--m-->
error error error

/SCOPE=scope_name
Specifies the level in the model hierarchy for this DETECT command only, with no effect on the SET SCOPE setting.

/SETUP=time
Used with /STROBE, describes the minimum time for which the logic signal at the detection point must remain stable prior to the strobe being asserted.

/SEVERITY = message_severity_level
Used with /MESSAGE_CONDITION. This qualifier sets the severity level of the messages DECSIM reports during a simulation that uses the detection points defined by the DETECT command being entered. There are four keywords that denote the degree of severity: INFORMATIONAL, WARNING, ERROR, and FATAL. Only one level of severity can be specified per DETECT command. All detection points declared by that particular DETECT command will output messages at that level.

By default, the messages are listed as informational.

The /SEVERITY qualifier is specified with the SET ERROR/ABORT command to stop simulation if timing violations occur.
/STATIC
/STATIC causes DECSIM to examine the detection points on the rising edge of the strobe. You can use /STROBE to establish aperiodic sampling or /EVERY to establish a periodic detection strobe. This is in contrast to /DYNAMIC, which activates the detector for the whole period of time that the strobe is asserted.

/STATIC is the default. See /DYNAMIC for level-sensitive strobes.

/STROBE=signalname
Specifies a signal that, when asserted high, causes the good machine to be compared to the faulty machine. /STROBE permits aperiodic sampling. When the strobe signal is asserted (logic 1), fault detection is declared for those faults that are visible at the detection point. U#2 is not considered to be a 1.

/THRESHOLD = integer
Establishes the number of times a fault must be detected before it is removed from the simulation. One threshold is used throughout the simulation; however, if another DETECT command with another threshold is entered, that threshold overrides the previous one.

By default, DECSIM removes a fault after the first detection.

/THRESHOLD is also used with /OSCILLATION. /OSCILLATION checks the number of times a node goes through a transition in a specified period of time and issues a message if the specified number of transitions is exceeded. /THRESHOLD is used to set the acceptable number of transitions.

/TOGGLE
Used in good simulation only. /TOGGLE checks to find which nodes are changing state. /TOGGLE detects state changes on the specified nodes. By default, it detects state changes to 0 and to 1. However, /COVERAGE overrides the default state changes, and /THRESHOLD overrides the default number of times the specified signals have to change in order to be detected. The default for /TOGGLE is /COVERAGE = (0,1)/THRESHOLD = 1.

/WINDOW=time
Specifies the minimum amount of time that a fault effect (discrepancy between the good machine and the faulty machine) must remain visible at a detection point before being detected. Fault effects that are visible for less than the amount of time specified with /WINDOW are not detected. If a strobe signal is specified, the window width must be less than the duration of the strobe signal assertion.

DESCRIPTION
The DETECT command simulates the probes of a tester, in that it determines the faults that can be detected at a certain location.

Detection points can be placed on structural memory cells and on vector outputs. Ranges of word subscripts and bit subscripts can be specified in memory detectors to place detect points on a subset of the words in an array or a subset of the bits in a word. For example: ARRAY_NAME[6]<14:10>.
The /NON_DETECTION, /POSSIBLE_DETECTION, and /FIRM_DETECTION qualifiers allow the user to specify what constitutes a fault detection. These qualifiers take a list of argument pairs that describes the valid good and faulty signal state combinations to be considered as possible or firm detections. If the state of the good machine differs from the state of the faulty machine, the good/fault pairs are checked against the list of state pairs specified with the qualifiers, and a detection is reported if a match exists.

The states that DECSIM supports are as follows:

- 0 – binary stable state
- 1 – binary stable state
- Z – high impedance stable state
- U – undefined state
- T0 – transition ending in a 0 state
- T1 – transition ending in a 1 state
- TZ – transition ending in a Z state
- TU – transition ending in a U state
- DC – don’t care state

T0, T1, TZ, and TU are used with the /DYNAMIC qualifier. See the description of this qualifier for more information. Also see the system variable descriptions of %POSS_DETECTED and %FIRM_DETECTED under %SYSTEM VARIABLES in this chapter.

For each DETECT command, an existing or new file may be named that reports the faults detected for the fault detectors described by the command. After each SET DETECT command, DECSIM lists both the number of detectors inserted by the command and the total number of detectors in the network.

ZYCAD Mode

In ZYCAD mode, the simulation results of detection points are evaluated at specific time intervals and saved. The format of the DETECT command is:

```
DETECT [/SCOPE=scopename ] /FROM=time /EVERY=time access_list
```

The /FROM qualifier specifies the time from which the good machine is compared with the fault machine(s).

The value of the /EVERY qualifier specifies the intervals, starting with the time specified in the /FROM qualifier, at which the good and faulty machines are compared.

One set of /FROM and /EVERY qualifiers is required for every DECSIM/ZYCAD fault run. If more than one DETECT command is used, you should specify the /FROM and /EVERY qualifiers on at least one. If the SIMULATE command is given without the /FROM and /EVERY qualifiers on at least one DETECT command, a warning message is issued and ZYCAD simulation does not occur.
The DETECT command without the /FROM and /EVERY qualifiers takes values from the last set used for the simulation.

Note: DETECT/EVERY differs from WATCH/EVERY and TRACE/EVERY. With DETECT, the action starts at the time specified with the /FROM qualifier. With WATCH and TRACE, action starts with the current simulator clock time.

For more information see:
DROP FAULT ZYCAD

EXAMPLES

SIM> SET DETECT/LOGNAME=Dictionary tp0, tp1, tp2
SIM> SET DETECT/STROBE=light street
SIM> DETECT/OSCILLATE/THRESHOLD = 2/EVERY = 1 *
%I-DTN, DETECT Name is DETECT2
Inserted 8 toggle or oscillation detection points, total of 8 in network
SIM> SIMULATE 3
Excessive events scheduled on the node: C0
Excessive events scheduled on the node: C0

EXAMPLE

The following example shows the output of DETECT. By default, this output is sent to the screen. /LOGNAME = logname sends the output to the specified log file. In the column labeled "Detect Activity," "Good" indicates that a signal transition in the good machine produced a fault detection. "Fault" indicates that a signal transition in the faulty machine produced a fault detection. The asterisk (*) indicates the first detection on each fault source.

<table>
<thead>
<tr>
<th>Time Detected</th>
<th>Location of Fault Origin</th>
<th>Location &amp; Good State of Detect Point</th>
<th>Faulty Detect Node</th>
<th>Activity Percent of Total Inserted</th>
</tr>
</thead>
<tbody>
<tr>
<td>135 ns</td>
<td>(ALPHA:OSA1)</td>
<td>R32.Q NOT 0</td>
<td>1</td>
<td>Good*</td>
</tr>
<tr>
<td>1234 ns</td>
<td>(FOOBAR:ISA0)</td>
<td>R32.QB NOT 1</td>
<td>0</td>
<td>Fault*</td>
</tr>
</tbody>
</table>
**EXAMPLE**

The following example illustrates the effects of the `/WINDOW` qualifier. If faults are visible (illustrated with *) at the detection point for at least the duration of the window, they are detected. No strobe signal exists, but timing violations (whose detection is dependent on the duration of fault visibility) are reported when they occur. The window is opened at the time the faults become visible; when the window closes, faults that remain visible are then detected. Note that the faults that did not remain visible are not classified as detected, and also that timing violations are reported for good signal changes that occur during the window, but not for similar faulty signal changes.

```
DETECT/WINDOW=w

Window  |<--w-->|  |<--w-->|  |<--w-->|  |<--w-->|  |<--w-->|
Faulty 1  ***********  ****--------------------------***********  +---
          0 | ________ |_______ |_______ |_______ |
          1 +--------+ +-------+ +-------+ +-------+
Fault-0  ___________________________ | (T) | __________ |
free detection?  Y | N | N | Y | N
              N = "No", Y = "Yes", T = "timing violation"
```

**EXAMPLE**

The following example illustrates the effects of the `/STROBE` or `/EVERY` qualifiers. Faults are detected if they are visible at the detection point at the time the strobe signal is asserted. Under all conditions, use of `/EVERY` or `/STROBE` achieves the same result; however, aperiodic test-point sampling may be specified by `/STROBE`, while `/EVERY` describes a fixed-frequency strobe. The strobe signal is typically a “pseudo-input” to the network, driven by a DEPOSIT or PATTERN input, to binary states only (0,1).

```
DETECT/FROM=0/EVERY=100

fault-free  1 +--------+
signal      0 ____________________________
faulty signal 1 +----------------------++---
               0 | ___________
strobed signal 1 +--------+ +-------+ +----
                 0 | __________
detection?    N  Y  N  N  N  N  N  N  N  N
              N = "No", Y = "Yes", T = "timing violation"
```
DETECT

EXAMPLE

The following example illustrates the effect of using /STROBE with /WINDOW. /WINDOW in conjunction with /STROBE causes the window to be opened at the time the strobe is asserted, rather than at the time of a good or faulty signal change. Note that timing violations are reported for good signal changes that occur during the window, but not for similar faulty signal changes.

DETECT/STROBE=sig/WINDOW=w

<table>
<thead>
<tr>
<th>window time</th>
<th>w-w-</th>
<th>w-w-</th>
<th>w-w-</th>
</tr>
</thead>
<tbody>
<tr>
<td>fault-free</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>signal</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>faulty signal</td>
<td>+-----+</td>
<td>(T)</td>
<td></td>
</tr>
<tr>
<td>strobed signal</td>
<td>+----+</td>
<td>+----+</td>
<td></td>
</tr>
<tr>
<td>detection?</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

N = "No", Y = "Yes", T = "timing violation"

EXAMPLE

The following example illustrates using /SETUP with /STROBE or /EVERY. Faults are detected if they are visible at the detection point at the time the strobe signal is asserted, and have been visible for the duration of the SETUP time given, 's.' Timing violation (detector strobe setup time violation) is reported if the visibility of the fault(s) changes during the setup time period. Timing violations are reported for good signal changes that occur during the setup time, but not for similar faulty signal changes.

DETECT/STROBE=sig/SETUP=s

<table>
<thead>
<tr>
<th>setup time</th>
<th>s-s-</th>
<th>s-s-</th>
<th>s-s-</th>
<th>s-s-</th>
<th>s-s-</th>
</tr>
</thead>
<tbody>
<tr>
<td>fault-free</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>signal</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>faulty signal</td>
<td>+-----+</td>
<td>(T)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strobe signal</td>
<td>+----+</td>
<td>+----+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>detection?</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

N = "No", Y = "Yes", T = "timing violation"
The following example illustrates using /STROBE, /SETUP, and /WINDOW, or using /EVERY, /SETUP, and /WINDOW. Faults are detected if they are visible at the detection point at the time the strobe signal is asserted, have been visible for the duration of the SETUP time given, and remain visible for the duration of the WINDOW specified. Note that timing violations are reported for good signal changes that occur during the setup or window time, but not for similar faulty signal changes.

DETECT/STROBE=sig/SETUP=s/WINDOW=w

<table>
<thead>
<tr>
<th>setup time</th>
<th>[s]</th>
<th>[s]</th>
<th>[s]</th>
<th>[s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>window time</td>
<td>[w]</td>
<td>[w]</td>
<td>[w]</td>
<td>[w]</td>
</tr>
<tr>
<td>fault-free signal</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>faulty signal</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strobe signal</td>
<td>1</td>
<td>++</td>
<td>++++</td>
<td>++++</td>
</tr>
<tr>
<td>detection?</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

N = "No", Y = "Yes", T = "timing violation"
CANCEL DETECT

CANCEL DETECT removes the specified fault detector or detectors from the network.

FORMAT

CANCEL DETECT [qualifier] detect_name

Default Qualifiers
None.

Optional Qualifiers
/ALL

restrictions
None.

ARGUMENTS
detect_name
Specifies a set of detectors identified with the name given by a previous DETECT/DTECTNAME command.

QUALIFIERS
/ALL
Removes all fault detectors from the network.

DESCRIPTION The CANCEL DETECT command is used to remove a detector or a set of detectors from the network. The detectors to be removed are specified by using the detect_name specified with DETECT/DTECTNAME=detect_name (or the name assigned to the set by default if no name was specified.)
DISABLE/ENABLE DETECT

The DISABLE/ENABLE DETECT commands temporarily disable and enable the detection of the specified set of signals.

FORMAT

DISABLE DETECT  [/qualifier] [detect_name]
ENABLE DETECT  [/qualifier] [detect_name]

Default Qualifiers
None.

Optional Qualifiers
/ALL

restrictions
None.

ARGUMENTS  
detect_name  
The name for the set of detectors that was specified with the /DETECTNAME qualifier on the DETECT command (or the default name of DETECTn assigned by DECSIM).

QUALIFIERS  
/ALL  
Specifies that all sets of detectors should be disabled or enabled.

DESCRIPTION  
The DISABLE DETECT command temporarily stops fault detection on the specified detectors. Detection resumes when the ENABLE DETECT command is entered.
SHOW DETECT

Shows the current status of fault detection.

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>SHOW DETECT  [/[qualifier] [detect_name,...]]</th>
</tr>
</thead>
</table>
| Default Qualifiers | /FAULT  
|               | /NOMAP  
|               | /NOSUMMARY  
|               | /OSCILLATION  
|               | /TOGGLE  |
| Optional Qualifiers | /NOFAULT  
|               | /INDEX  
|               | /MAP  
|               | /NOOSCILLATION  
|               | /NOTOGGLE  
|               | /SUMMARY  |

restrictions

None.

ARGUMENTS
detect_name

Specifies the detectors to be displayed as those identified by the /DETECTNAME qualifier on a DETECT command.

QUALIFIERS

/FAULT
/NOFAULT

Prints information on all fault detectors in the network or on those fault detectors specified in the detect_name. /SUMMARY can also be specified to print a less detailed summary of detections for each fault detector.

/NOFAULT is the default.

/INDEX

Abbreviates the display to list only the detector names with their corresponding numeric identifiers. Numeric identifiers are assigned to detectors with the command DETECT/INDEX.

/MAP
/NOMAP

Prints a function map of the detection criteria established for all detection points placed in the network. See the command description of DETECT for an explanation of the nine valid states that may be specified with DETECT/FIRM_DETECTION.
SHOW DETECT

/NOMAP is the default.

/OSCILLATION
/NOOSCILLATION
Prints information on all oscillation detectors in the network or on those detectors specified in the detect_name. /SUMMARY can also be specified to print a less detailed summary for each oscillation detector.

/NOOSCILLATION is the default.

/SUMMARY
/NOSUMMARY
Prints only the total tallies for each detector in the network. The type of detector for which you want information can be specified with /FAULT, /TOGGLE, or /OSCILLATION. Specific detectors can be specified in the detect_name.

/NOSUMMARY is the default.

/TOGGLE
/NOTOGGLE
Prints information on all toggle detectors in the network or on those detectors specified in the detect_name. /SUMMARY can also be specified to print a less detailed summary for each toggle detector.

/NOTOGGLE is the default.

**DESCRIPTION**

The SHOW DETECT command summarizes the current status of fault detection by:

- Identifying the detect_name associated with the current set of detectors. For example, in the display heading of the following example, "Summary of fault detectors -DETECT1," DETECT1 is the detect_name.

- Listing the location and numeric identifier of each detector that has been inserted.

- Showing the conditions associated with the detectors, as determined by the qualifiers specified with the DETECT command that was used to insert them.

- Reporting the number of undetected faults.

- Reporting the percentage of the total number of inserted faults that are left in the network.

- Reporting the average number of detections per each fault.
SHOW DETECT

EXAMPLE

SIM> show detect
Summary of fault detectors -DETECT1

Detector-Location: Detections: Status: Index:
                (Firm/Poss)
---------------------------------------------------------------------------------
C2   Firm:1; Poss:0 ENABLED (3)
/STATIC /WINDOW:None /MIN_PULSE_WIDTH:None /CONTINUOUS_VISIBILITY
/REPORT:(SETUP:(REFERENCE NOCONCURRENT),
     WINDOW:(REFERENCE NOCONCURRENT),
     PULSE:(REFERENCE NOCONCURRENT))
/SEVERITY:INFORMATIONAL /LOG
SUM2  Firm:1; Poss:0 ENABLED (2)
/STATIC /WINDOW:None /MIN_PULSE_WIDTH:None /CONTINUOUS_VISIBILITY
/REPORT:(SETUP:(REFERENCE NOCONCURRENT),
     WINDOW:(REFERENCE NOCONCURRENT),
     PULSE:(REFERENCE NOCONCURRENT))
/SEVERITY:INFORMATIONAL /LOG
SUM1  Firm:3; Poss:0 ENABLED (1)
/STATIC /WINDOW:None /MIN_PULSE_WIDTH:None /CONTINUOUS_VISIBILITY
/REPORT:(SETUP:(REFERENCE NOCONCURRENT),
     WINDOW:(REFERENCE NOCONCURRENT),
     PULSE:(REFERENCE NOCONCURRENT))
/SEVERITY:INFORMATIONAL /LOG
---------------------------------------------------------------------------------
Total =  5 /  0
Total of  19 Undetected Faults (  79.166% of Faults Inserted).
FIRM Detections per Fault Average  1.000

SIM> SHOW DETECT/INDEX
Summary of fault detectors -DETECT2

Detector-Location: Index:
---------------------------
SUM2          (8)
SUM1          (7)
C2            (5)

SIM> SHOW DETECT/MAP
Summary of fault detectors -DETECT1

Detector-Location: Detections: Status: Index:
                (Firm/Poss)
---------------------------------------------------------------------------------
C2   Firm:1; Poss:0 ENABLED (3)
/STATIC /WINDOW:None /MIN_PULSE_WIDTH:None /CONTINUOUS_VISIBILITY
/REPORT:(SETUP:(REFERENCE NOCONCURRENT),
     WINDOW:(REFERENCE NOCONCURRENT),
     PULSE:(REFERENCE NOCONCURRENT))
/SEVERITY:INFORMATIONAL /LOG

5-54
### FAULT-FREE STATE

<table>
<thead>
<tr>
<th>F</th>
<th>U</th>
<th>T</th>
<th>T1</th>
<th>T2</th>
<th>TU</th>
<th>DC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>FD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Legend:**
- F - FIRM Detection
- P - POSSIBLE Detection
- M - MESSAGE output
- D - DROP Detected Fault

Total detections:
- Total detections: 5
- Undetected faults: 0

**Summary:**
- Total of 19 Undetected Faults (79.166% of faults inserted).
- FIRM Detections per Fault Average: 1.000
**DROP**

Controls when faults are dropped from the network after they have been detected.

**FORMAT**

[SET] DROP  \textit{integer}

SHOW DROP

**Default Qualifiers**

\textit{None}.

**Optional Qualifiers**

\textit{None}.

**restrictions**

- The DROP command is used in fault simulation only.

**ARGUMENTS**  \textit{integer}

The percentage of the remaining fault sources that must be detected before a fault dropping pass is initiated. This value must be between 1 and 100.

**DESCRIPTION**

The SET DROP command allows the user to control the mechanism by which faults are purged from the simulator. Detected faults are not dropped immediately and continue to be simulated, but they are placed on a drop list. If the number of fault sources on the drop list meets or exceeds the specified percentage of fault sources remaining in the simulation, a dropping pass is initiated, and all faults on the list are dropped at the same time.

Low DROP values allow the simulator to operate more efficiently by letting it terminate the simulation of detected faults at earlier times. Larger values give the user more information about the number of times a fault is detected.

SHOW DROP lists the current drop threshold and gives the number of times that faults have been dropped, the number of fault sources and effects dropped, and the number of fault effects still in the network.

For more information see:

DETECT FAULT

**EXAMPLE**

\texttt{SIM> DROP 50}

This setting says that 50% of the fault sources remaining in simulation must be detected before a dropping pass is initiated.
EDIT

Allows you to edit a file from the interactive command language.

FORMAT

EDIT [/qualifier] file_specification

Default Qualifiers
/EDT

Optional Qualifiers
/LSE
/TPU
/EMACS

restrictions

- EMACS is not supported and is available only to users who have set up an EMACS environment.

QUALIFIERS

/EDT
Invokes the EDT editor. This is the default.

/LSE
Invokes the LSE editor.

/TPU
Invokes the TPU editor.

/EMACS
Invokes the EMACS editor

DESCRIPTION

EDIT invokes an editor and places the specified file in it. When you exit from the editor, you are returned to the SIM> prompt.

EDIT uses the DCL EDIT command to invoke the default editor, EDT. You can invoke a different editor by using a qualifier. You can also redefine the DCL EDIT command in your login.com file by using DEFINE DECISIM$EDIT CALLABLEEDITORNAME where editorname can be LSE, TPU, or EMACS.
EXAMPLE

SIM> type sample.file
Good morning
SIM> EDIT sample.file

! The editing, which occurs at this point, is not shown.

SIM> TYPE sample.file
Good afternoon
EDITCOMMANDMACRO (Command Macro)

Allows you to edit a command macro while in the interactive command language mode.

FORMAT

EDITCOMMANDMACRO 'macro_name'

Default Qualifiers
None.

Optional Qualifiers
None.

restrictions
None.

DESCRIPTION

The EDITCOMMANDMACRO invokes an editor and places the specified command macro definition in it. When you exit from the editor, you are returned to the SIM> prompt.

The EDITCOMMANDMACRO can be abbreviated to 5 characters: EDITC.

The macro_name should be enclosed in single quotation marks to avoid expansion.

EXAMPLE

SIM> show macro/command 'greeting'
    SET MACRO /COMMAND GREETING [ $TEXT ] = PRINT '$TEXT' $ENDMACRO;
SIM> edita 'greeting'
    SET MACRO /COMMAND GREETING [ $TEXT ] = PRINT '$TEXT' $ENDMACRO;
%I-CLO, Closed: LVS$: [RANDALL.WORKDECSEM] EDITMACRO.TMP;31
! The editing, which occurs at this point, is not shown.
SIM> show macro/command 'greeting'
    SET MACRO /COMMAND GREETING [ $TEXT ] = PRINT '$TEXT, John' $ENDMACRO;

EDITCOMMANDMACRO Macro Definition

SET MACRO /COMMAND /ABBREV:5 EDITCOMMANDMACRO [ $NAME ] =
    SET LOG/RESP/NOCOMMA/NOHEAD/NOTRAC/NOECHO
    /NOCOMME/LOG=EDITMACRO EDITMACRO.TMP
    SHOW MACRO/COMMA '$NAME'
    CANCEL LOG EDITMACRO
    CANCEL MACRO/COMMA '$NAME'
    EDIT EDITMACRO.TMP
    @EDITMACRO.TMP
$ENDMACRO;
EDITTEXTMACRO (Command Macro)

Allows you to edit a text macro while in the interactive command language mode.

FORMAT  EDITTEXTMACRO 'macro_name'

Default Qualifiers
None.

Optional Qualifiers
None.

restrictions
None.

DESCRIPTION
The EDITTEXTMACRO invokes an editor and places the specified macro in it. When you exit from the editor, you are returned to the SIM> prompt.

EDITTEXTMACRO can be abbreviated to 5 characters: EDITT.

The macro_name should be enclosed in single quotation marks to avoid expansion.

EXAMPLE

SIM> show macro 'greet'
    SET MACRO /TEXT GREET = print 'Hello' $ENDMACRO;
SIM> edit 'greet'
    SET MACRO /TEXT GREET = print 'Hello' $ENDMACRO;
    %L-CLO, closed: LVSS:[RANDALL WORKDECISM]EDITMACRO.TMP;37
    ! The editing, which occurs at this point, is not shown.
SIM> show macro 'greet'
    SET MACRO /TEXT GREET = print 'Good Morning' $ENDMACRO;
SIM> exit

EDITTEXTMACRO Macro Definition

SET MACRO /COMMAND /ABBREV:5 EDITTEXTMACRO [ $NAME ] =
 SET LOG/RESP/NOCOMMA/NOHEAD/NOTRAC/NOECHO
 /NOCOMME/LOG=EDITMACRO EDITMACRO.TMP
 SHOW MACRO/TEXT '$NAME'
 CANCEL LOG EDITMACRO
 CANCEL MACRO/TEXT '$NAME'
 EDIT MACRO/TEXT '$NAME'
 @EDITMACRO.TMP
 $ENDMACRO;

5–60
ERROR
Specifies the disposition and handling of error messages.

FORMAT
[SET] ERROR [ /qualifier... ]
SHOW ERROR
CANCEL ERROR /qualifier

Default Qualifiers
/ABORT=FATAL
/DISPLAY=(NOFAC,SEV,ID,TEXT,NOTIME)
/LIMIT=INFINITY
/NOBREAK
/NOVERBOSE
/REPORT=ALL

Optional Qualifiers
/ABORT=severity
/BREAK
/DISPLAY=(Qualifier,...)
/LIMIT=integer
/REPORT=severity
/VERBOSE

restrictions
None.

QUALIFIERS
/ABORT=severity
Sets the minimum severity level of DECSIM messages that stop further execution. It is recommended that in large noninteractive jobs you lower the abort level from the default of FATAL to at least ERROR so that the job will not continue after encountering a condition such as "network didn't compile," "file not found," or "syntax error." If the specified abort level is below the report level, the abort level is set equal to the report level, and no warning is given.

/BREAK
/NOBREAK
Suspends simulation (that is, executes the SET BREAK command) when any message is reported. This qualifier is intended for interactive debugging.

/DISPLAY=qualifier
/DISPLAY=(qualifier,...)
Defines what parts of an error message are printed. The parts of the error message are specified with the following arguments:
TIME
NOTIME

Specifies that messages be timestamped with the
current simulation clock when the last entered
command was SIMULATE.

FACILITY
NOFACILITY

This is usually DECSIM, except for a few system
messages.

SEVERITY
NOSEVERITY

Displays I (informational), W (warning), E (error), or F
(fatal).

IDENTIFICATION
NOIDENTIFICATION

Displays the ID field of the message.

TEXT
NOTEXT

Displays the text of the error message.

/LIMIT=errorlimit
/LIMIT=INFINITY

Sets the number of messages DECSIM can print before execution is
aborted. The default error limit is INFINITY.

/REPORT=severity

Sets the minimum severity level of messages that DECSIM reports.
Initially, the report level is ALL, meaning messages of all severities are
reported. If the specified report level is above the abort level, the report
level is set equal to the abort level, and no warning is given.

DESCRIPTION

Every message includes a severity level that describes the seriousness of
the condition being reported, as follows:

• Informational – Informative message about the last entered command.

• Warning – The last entered command encountered an incongruity, but
simulation can continue with DECSIM making some assumptions.

• Error – The last entered command encountered a condition that
prevents the command from finishing. Examples of error conditions
include "syntax error," and "file not found."

• Fatal – DECSIM cannot continue execution.

The severity parameter of the /REPORT and /ABORT qualifiers specifies
one of the above levels, or ALL to mean all of them.

SHOW ERROR shows the status of the error feature.

CANCEL ERROR/ABORT returns the abort level to the level that
was in effect prior to the current abort level. The SET and CANCEL
ERROR/ABORT commands can be used to temporarily set an abort level
for a sequence of commands, as in an indirect file, and then return to the
previous setting without knowing what that setting was.

CANCEL ERROR/REPORT is similar to CANCEL ERROR/ABORT, but
returns the previous report severity level.

Both CANCEL commands have the ability to remember the previous 8
settings. More than 8 SET levels can be entered, but only the last 8 can
be returned to; then the initial default is used.
EXAMPLES

SIM> SIM
%E-NNL, Network not loaded
SIM> ERROR/DISPLAY=FAC
SIM> SIM
%DECSIM-E-NNL, Network not loaded
SIM> ERROR/REPORT=FATAL
SIM> SIM
SIM> SHOW ERROR
    ERROR /REPORT=FATAL /LIMIT=NONE /ABORT=FATAL /NoBREAK
    /DISPLAY=(FAC, SEV, ID, TEXT, NoTIME)
    Errors detected: 2
SIM> CANCEL ERROR/REPORT
%DECSIM-I-CANREPORT, Report level returned to ALL
SIM> ERROR/BREAK/DISPLAY=NoFAC
SIM> SIM
%E-NNL, Network not loaded
    Break after reporting error message
SIM>BREAK> RESUME
SIM> ERROR/NoBREAK/LIMIT=1
SIM> SIM
%E-NNL, Network not loaded
%F, error limit reached
$
EVALUATE (EXPRESSION EVALUATION)

Calculates the value of an expression or assigns a value to a state or memory location. The EVALUATE command will not assign memories in ZYCAD mode.

**FORMAT**

EVALUATE expression \[ ,... \]
EVALUATE target \[ = expression ,... \]

**Default Qualifiers**

None.

**Optional Qualifiers**

None.

**restrictions**

- The EVALUATE command assigns values to states and memory arrays only.
- This command does not assign values to memory locations in ZYCAD mode.

**DESCRIPTION**

The EVALUATE command takes a valid expression, calculates it, and either prints the result on the terminal or, if a target state is given, stores the value in the state.

The expression consists of operators and operands. The assignment target may be any valid variable, memory location, or signal name. Operators may be any of those described in Appendix D. Operands can be any valid signal name, internal behavior state, memory location, or defined command variable. EVALUATE is therefore particularly useful in debugging logic and models, and in generating test pattern stimuli.

If you do not define the target of an EVALUATE command assignment, DECSIM uses the implicit STATE command, which ignores bit and word subscripts.

If you do not define an operand variable, the default value is 0.

**Note:** Although Behavior STATES and network signal names may be abbreviated, STATE variables defined by the command language may not.

For more information see:

CALL expressions operators
DEPOSIT numbers PATTERN
SET MACRO
EVALUATE (EXPRESSION EVALUATION)

EXAMPLES

SIM> EVALUATE A AND B OR C
   1
SIM> EVALUATE 128*10#8
   1024
SIM> STATE VAR1
SIM> EV VAR1 = 2
SIM> SHOW STATE
   VAR1 = 2
SIM> EV VAR1 = VAR1 + 1, VAR1
   3
SIM> EV V[2]<3> = 1
SIM> EV V[J]<J+1:J-1> = 7
SIM> FOR I FROM 0 TO 7 DO
   MORE> EV V[I]<7-I> = 1

Control Flow Tracing

SIM> EV X=1
   [ EVALUATE assignment: X = 1$2 ]
SIM>
EXAMINE

 Allows you to inspect the state of signals and labels, as well as the contents of memory arrays and behavior model states.

FORMAT

EXAMINE [/qualifier...] access_list

[(machine_descriptor)]

Default Qualifiers
/NOSYNONYM
/RADIX=current_radix
/SCOPE=current_scope
/SEPARATOR="M^J/0/1/Z/U

Optional Qualifiers
/0/1/Z/U
/ALL_FAULTS
/BRIEF
/EXACT
/FANIN
/FANOUT
/FAULTS/ORIGINS
/FULL
/INVISIBLE
/LABELS
/PARAMETERS
/RADIX=integer
/SCHEDULED
/SCOPE=scope_name
/SEPARATOR="Text"
/SYNONYM

restrictions

None.

ARGUMENTS

access_list

A list, separated by commas, of one or more signal names, behavior internal states, memory arrays, or state variables. The access list may contain wildcards. Double quotation marks are necessary for character matching.

Machine descriptors can be used to examine signals in faulty machines. The syntax is: EXAMINE signal_name [machine_descriptor]. It is not necessary to use SET CONTEXT to change the machine to which the EXAMINE command applies. See Section 4.3.8 for more information on machine descriptors.
QUALIFIERS

\texttt{/0 /1 /Z /U}

Print the signal name (and value) only if the current value matches the given qualifier. This may be used in conjunction with a list of names, or you can select from names in the current scope that match the value specified with the qualifier.

\texttt{/ALL_FAULTS}

Reports the value of all faulty machines as specified with the CONTEXT command, whether the faults are visible (different from the good machine) or invisible (the same as the good machine).

\texttt{/BRIEF}

Prints the name only, or if you are examining a label, prints the label only. Useful for symbol searching using wildcards.

\texttt{/EXACT}

Uses the form of name as you gave it in the access list specification (even when abbreviated), rather than translating it to the preferred name. Using a wildcard in the access_list is an exception. The names used will be the names as given in the model.

\texttt{/FANIN}

\texttt{/FANOUT}

The \texttt{/FANOUT} and \texttt{/FANIN} qualifiers have different results depending on whether the indicated signal is an output, input, or MOS-node.

\texttt{/FANOUT output--signal}

Lists the names of the inputs that the specified output--signal drives. If it drives none, it indicates it is 'unconnected'. Outputs may drive special data--convertor elements (called 'splits' and 'merges') which interface vector--format signals to scalar--format signals. These are indicated in the input--list by a 'MERGE' or 'SPLIT' keyword, followed by the signal driving the split or merge and by the name of the input(s) driven by the split or merge.

\texttt{DISPLAY FORMAT:}

\begin{verbatim}
output-name ===> input-1-name, input-2-name, ...
MERGE: merge-input-name -> input-1-driven-by-merge,
input-2-driven-by-merge, ...
SPLIT: split-input-name -> input-1-driven-by-split,
input-2-driven-by-split, ...
\end{verbatim}

\texttt{/FANOUT input--signal}

Since input--signals do not drive anything, this command merely indicates that the input--signal is an element--input.

\texttt{/FANOUT MOS--node}

Lists the names of the inputs that the specified MOS--node drives. The format for this display is identical to that for \texttt{/FANOUT} for an output--signal. Note that the signals displayed will be inputs only, not any MOSFET Sources or Drains that connect to the MOS--node. To display the MOSFET Source or Drains, use the \texttt{/FANIN} qualifier.
/FANIN input-signal

Displays the output-signal which drives the specified input and the value of that output-signal. Inputs may be driven by special data-convertor elements (called 'splits' and 'merges') which interface vector-format signals to scalar-format signals. These are indicated by a 'MERGE' or 'SPLIT' keyword, followed by the output signal(s) driving the split or merge.

DISPLAY FORMATS:

input-name <= output-name = output-value
input-name <= SPLIT = split-output-value
output-driving-split = output-value
input-name <= MERGE = merge-output-value
output-1-driving-merge = output-value-1
& output-2-driving-merge = output-value-2

/FANIN output-signal

Lists all the inputs to the primitive element (memory, behavior, etc.) which has the specified output. For each input, it displays the output-signal which drives the input and the value of that output-signal (in the same format as /FANIN for an input-signal).

/FANIN MOS-node

Displays all the MOSFET primitives which connect to this MOS-node through a Source or Drain connection. For each MOSFET, it lists its gate-terminal name, its Source or Drain name, and whether it is a pullup, pulldown, or transmission-gate. Parameter information for the MOSFET can be displayed by combining /FANIN and /PARAMETERS (see the /PARAMETERS qualifier).

/FAULTS
/ORIGINS

Shows fault effects or fault sources associated with the examined signals.

/FULL

Prints the full hierarchical name, when EXAMINE reports the name, from the currently defined SCOPE down to the primitive level.

/INVISIBLE

After you have used the SET CONTEXT command, reports the name, value, and machine descriptor of a faulty machine that is present but is now the same as the good machine.

/LABELS

Specifies that the items to be examined are labels, not signal names. This qualifier can also be used to get routine names from behavior models.

/PARAMETERS

Specifies that for the specified signal name, you examine the parameters instead of the state. When applied to a memory array, provides the following information:

- Memory array name with word and bit subscripts. The word subscripts show min and max addresses of the array. The bit subscripts show data path width.
- Default memory state.
• Status of each port in the memory, including whether it is READ or WRITE, the FUNCTION word, DISABLE state, and all delay parameters.

When applied to a MOS model, provides the following information:
• The state of the node.
• The attributes associated with the node, including capacitance, threshold voltages, timeout, and charge sharing.

Using the /FANIN and /PARAMETERS qualifiers together for MOS primitives displays transistor data in addition to the usual information shown by each qualifier. It shows the type of transistor (N, P, D, DI, R, or Z), its width/length ratio, how it is connected in the network (as a pull-up, pull-down, etc), and the various resistances associated with it.

The following example illustrates the format of the command. You can see that resistance is calculated for the stack as a whole, and also for the individual gates. The three resistances for N, P, R, or Z transistors are displayed on single lines, and the six resistances for D and DI transistors are displayed on consecutive lines.

```
SIM> examine/parameters/fanin NODE1
NODE1 = U
pF 7.202E-2 0.8 
NODE1 <===
Thr_H Thr_L Timeout Chg_Shr
  0.2     Infinity   yes
RStatic   RDyn H RDyn L
RStat Max  RDyn H Max RDyn L Max
Modeled as W/L Type

Pullup:
  4.642E5 6.152E4
  9.212E5 1.237E5
---
/ VDD = U

Pulldown:
  3.665E4
  %NET.NAND_IN1 = U N 6.667
  %NET.NAND_IN2 = U N 6.667
/ VSS = U

Xmission:
  6.106E4 1.756E4
  %NET.XM_IN = U N 2.0
/ OUT = U
```

/RADIX=integer
Specifies radix.

/SCHEDULED
Prints the signal name, value, and time and state of next transition for the signals scheduled to change.

/SCOPE= scope_name
Specifies the SCOPE setting for this EXamine command only. This qualifier does not affect the SET SCOPE command setting.

/SEPARATOR= 'text'
Formats the outputted information with text in between each item. The separator default is carriage return/linefeed.

5–69
/SYNONYM
/NOSYNONYM
Displays a list of all synonyms of the signals. The last name in the
list is the preferred name. Pin numbers are treated as synonyms,
and are labeled [pin]. If there are no synonyms, the qualifier displays
[NoSYNONYM], followed by the preferred name.

When applied to vector signals, /SYNONYM reports only synonyms with
matching subscripts.

/SYNONYM does not apply to behavior synonyms.
/NOSYNONYM is the default.

DESCRIPTION
The EXAMINE command allows you to inspect the state of signals and
labels, as well as the contents of behavior model states and memory
arrays. You can also examine the state of signals in faulty machines by
including a machine descriptor after the signal name on the command line.

The abbreviation for EXAMINE is E.

Qualifiers apply only to the command they modify.

Wildcards are permitted.

By using the SET SCOPE command before using the EXAMINE command,
you can avoid writing the complete hierarchy in the access list.

To examine local state variables in a behavior model, prefix the state name
with the routine name and a dot. For example, to examine local state Z in
routine R, type EXAMINE R.Z.

Some of the EXAMINE qualifiers have been enhanced to give additional
information for MOS simulations. The combination of /FANIN and
/PARAMETER give more information about MOS than when used
separately.

For more information see:

DEPOSIT  PRINT
EVALUATE  RADIX
MODIFY    SCOPE
           WILDCARD
EXAMPLES

SIM> E/EXACT 0,0,0,0
C = U
CL = U
CLK = U

SIM> E/SCOPe=e1 out, out1, e3.cl.int
Scope = E1, Model = CHIP_II, Revision = 0
OUT = U
OUT1 = U
%NET.E3.C1.INT = U

SIM> EXAMINE/SYNONYM INT 1
INT 1 := INT_SYN 1 = U

SIM> EXAMINE/SYNONYM G2.1
G2.1 (pin) := output1 = U

SIM> E/FANOUT 01, 0,0, 0,1, 1,0
CLK = U
OUT1 = U
OUT2 = U
CLK ==> E1.IN, E4.IN
OUT1 ==> E2.IN, E3.IN
OUT2 ==> (Unconnected)

SIM> E/FANIN IN1
IN1 = U
IN1 <== (Primary Input)

E4.C1.G1.IN

SIM> E/BRIEF/PARAMETER out1, ooc, ooe, oot
OUT1
0 1 1 0 0 U 0 1 U 1 U
0 0 0 0 0 0 0
OOC
0 0 U 0 0 Z 0 Z U Z
2 3 2 3 3 2
OOE
1 U 1 Z U U Z Z 1 1 Z
4 5 4 5 4 4
OOT
0 1 1 0 0 U 0 1 U 1 U 0 Z 0 U 0 Z 1 1 Z
6 7 6 9 9 7 10 8 9 9 11 10 11

SIM> E/O *
CLK = 0
DRIVER = 0
OOC = 0
OOT = 0
OUT1 = 0
OUT3 = 0
OUT4 = 0
The following example shows the use of machine descriptors with EXAMINE. Signal B1 is a primary input and S2 is a primary output.
EXIT and QUIT

Terminate DECSIM execution and return you to the VMS prompt. EXIT is a final exit, whereas QUIT allows you to resume running DECSIM, as long as you do not disturb the memory image.

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>EXIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUIT</td>
<td></td>
</tr>
</tbody>
</table>

**Default Qualifiers**

*None.*

**Optional Qualifiers**

*None.*

**restrictions**

- There are only a limited number of VMS commands that can be executed after a QUIT command without disturbing the memory image of the suspended DECSIM session. See DESCRIPTION below.

**DESCRIPTION**

When executed interactively or from an indirect file, EXIT closes all output files (whether currently being written or not) and releases all allocated memory.

The QUIT command places your terminal at the VMS prompt, where you can invoke VMS commands that do not disturb the memory image, such as SHOW TIME. With the VMS CONTINUE command, you can resume your DECSIM session where you left off, whether QUIT was executed interactively or from an indirect file.

DECSIM warns you if a QUIT command is executed from an indirect file, or if output files are being written at the time of execution.

**Caution:** To save partially written output files, you must immediately use the CONTINUE command; otherwise the files are lost.

Normally, EXIT and QUIT are the recommended ways to exit from command mode to the VMS prompt. For emergency halts, use CTRL/C.
EXIT and QUIT

VMS commands to use between QUIT and CONTINUE

The following commands do not destroy the memory image:

VAX/VMS

$= $READ
$ALLOCATE $RECALL
$ASSIGN $RETURN
$ATTACH $SET CONTROL
$CALL $SET DEFAULT
$CANCEL $SET KEY
$CLOSE $SET ON
$CONNECT $SET OUTPUT_RATE
$CONTINUE $SET PROMPT
$CREATE/LOGICAL_NAME_TABLE $SET PROTECTION/DEFAULT
$DEALLOCATE $SET SYMBOL
$DEASSIGN $SET UIC
$DEBUG $SET VERIFY
$DECK $SHOW DEFAULT
$DEFINE $SHOW KEY
$DEFINE/KEY $SHOW PROTECTION
$DELETE/KEY $SHOW QUOTA
$DELETE/Symbol $SHOW STATUS
$DEPOSIT $SHOW SYMBOL
$DISCONNECT $SHOW TIME
$EOD $SHOW TRANSLATION
$EXAMINE $SPAWN
$GOSUB $STOP
$GOTO $WAIT
$IF $WRITE
$INQUIRE $OPEN
$ON

The above list applies to VAX/VMS Version 4.6.

For more information see:

CONTINUE (VMS level command) SET INDIRECT
CTRL/C VMS_COMMAND
LOG

EXAMPLES

SIM> EXIT
See ya later

SIM> QUIT
*I-QCONT, Type CONTINUE to return to DECSIM

$ SET DEF [-]
$ CONT
DECSIM: Welcome back!
FAULT

Describes the type and location of faults to be inserted in a network. Faults can be inserted in structural, memory and MOS models, but not in behavior or RealChip models.

The FAULT variants prefixed with SET and SHOW are described in the following sections.

FORMAT

[SET] FAULT [/qualifier...] access_list

Default Qualifiers
/NOPHANTOMS
/NOSCALAR
/NUMBER_OF_PASSES=1
/OUTPUT
/PASS=1
/PERCENT=100
/STUCK_AT=(0,1)
/THRESHOLD=1

Optional Qualifiers
/EXACT
/EXCEPT=(exception_list)
/INPUT
/LIMIT=integer
/NOLIMIT
/NOTHRESHOLD
/NUMBER_OF_PASSES=integer
/PASS=integer
/PERCENT=integer
/PHANTOMS
/SCALAR
/SCOPE=scope_name
/STUCK_AT=(state_list)
/THRESHOLD=integer

ZYCAD–specific Qualifiers
/FE_FAULTS_PER_PASS=integer

restrictions

• You cannot insert faults into behavior or RealChip models.

ARGUMENTS

access_list

The access list may include signal names or memory cell names. The full hierarchical name of the signal is required unless scope has been set to the appropriate level of the hierarchy. The wildcard * is allowed in signal names but not in memory cell names.
Word subscripts are required in memory cell names to indicate which cell or range of cells will be faulted. For example, `FAULT/STUCK_AT=(0) memblock[1]` inserts a single fault in a single memory cell, `memblock[1]`, while `FAULT/STUCK_AT=(0) memblock[0:1]` inserts a single fault in each memory cell, `memblock[0]` and `memblock[1]`. If no wordsubscripts are specified for a memory cell, no faults will be inserted.

Bit subscripts are not required in memory cell names unless you want to restrict fault insertion to a particular bit or range of bits. For example, `FAULT/STUCK_AT=(0) memblock[1]<3:0>` inserts a single fault on the four least significant bits of `memblock[1]`; the other bits of the cell are not faulted.

To insert faults on individual bits in a memory cell, use the `/SCALAR` qualifier. For example, `FAULT/SCALAR/STUCK_AT=(0) memblock[1]<3:0>` inserts a single fault on each of the four least significant bits of `memblock[1]` (a total of 4 faults). See the description of the `/SCALAR` qualifier below for more details.

**state_list**
A list of logic states; consists of 0's or 1's.

**exception_list**
A list of exceptions to the signals or elements in the access list. These exceptions are not assigned faults.

---

**QUALIFIERS**

`/EXACT`
Designates the form of the signal name to be displayed by the SHOW FAULT command as the name that you specified in the access list. By default DECSIM translates the name into the preferred name. With this qualifier, DECSIM displays any synonyms you have specified. See Section 4.3.7.4 for information about preferred names. See Section 2.4.3 and Section 3.6.4 for more information about the SYNONYM declaration.

`/EXCEPT=(exception_list)`
Specifies the signals in the access_list that are not to be assigned faults. The access_list of the FAULT command may be any list of signals, but is often a wildcard or wildcards.

Scope can be defined by specifying the levels of hierarchy in descending order. The levels are delimited by a period (.). Wildcards can also be used in the exception list. For example: `/EXCEPT = (%NET.D1.C1.B1.*,A*)`

`/FE_FAULTS_PER_PASS=integer`
A ZYCAD–specific qualifier used with the SET ZYCAD/FE command while simulating on the ZYCAD LE/FE, the concurrent fault evaluator (see the ZYCAD command). It allows you to limit the number of fault sources inserted in the network during each concurrent simulation pass on the fault evaluator. This number is lowered automatically if the FE hardware runs out of memory for fault effects and "stumbles". Each time the FE stumbles, it halves the faults–per–pass value until it can complete the simulation. By specifying a faults–per–pass value, you can tune your simulation around the ZYCAD hardware memory limitations and avoid stumbling. The default is the maximum, which is 16383 faults per pass.
/INPUT
/OUTPUT
Inserts fault sources on all inputs or outputs of the elements or signals in the access list. /OUTPUT is the default.

In MOS models, FAULT/INPUT inserts stuck-at faults on transistor gate input terminals. /OUTPUT inserts stuck-at faults on MOS nodes.

Note: If you use the /INPUT qualifier on a signal preferred name rather than on a signal designated with a label.pin name, you create an output fault on that signal.

/LIMIT = integer
/NOLIMIT
A fault may remain undetected even though it is causing a large number of fault effects in the network. /LIMIT helps to improve system performance by specifying a maximum number of nodes on which a faulty machine can have a differing value from the good machine. If this limit is exceeded by a fault, DECSIM removes the fault from the simulation and informs you of the fault removal in the default output file.

By default, for networks with 2,000 or more gate-level primitives or MOS primitives, /LIMIT is set to a number that is 25% of the number of primitives contained in that network.

For networks with less than 2,000 gates, the default is /NOLIMIT.
/NOLIMIT specifies no limitation on the number of fault effects a fault can have.

To obtain a list of the removed faults, use SHOW FAULT/REMOVED.

/NUMBER_OF_PASSES = integer
Used with the /PASS qualifier, the /NUMBER_OF_PASSES qualifier divides as equally as possible the total of the inserted faults into subgroups called passes. Faults are assigned to passes by the order of their locations as specified in the access list. The specified integer determines the number of passes; the default number of passes is one.

/PASS = integer
Used with the /NUMBER_OF_PASSES qualifier, the /PASS qualifier specifies which pass is used in the fault simulation. A pass is a subgroup of the faults in the access list. Faults are assigned to passes in the order of their placement in the access list with each pass assigned an equal share of faults.

When the SET FAULT/NUMBER_OF_PASSES = integer/PASS = integer command is executed, the faults that are not assigned to the specified pass are canceled and can be seen by entering the SHOW FAULT/CANCELLED command. For example:

SIM> SET FAULT/OUTPUT/STUCK_AT = (1) /NUMBER_OF_PASSES = 2/PASS = 2 a, b, c, d
This is pass number 2 out of 2 passes.
There are 2 active fault sources in this pass.
SHOW FAULT/UNDETECTED/CANCELLED
!
  Total faults inserted = 2
  Fault effect limit = NO LIMIT
  Executing pass number 2 out of a total of 2 passes
  Faults detected = 0  ( 0.000%)
    Firm = 0  ( 0.000%)
    Possible = 0  ( 0.000%)
  Faults removed = 0  ( 0.000%)

  Type       OUTPUT  INPUT  MEMORY  TOTAL
  ---------------  ------  ------  ------  ------
    S1  2  0  0  2

! Undetected Faults
--------------------------------------------------------------------------------
Fault/output/stuck_at:(1) C
Fault/output/stuck_at:(1) D

! Cancelled Faults
--------------------------------------------------------------------------------
Fault/output/stuck_at:(1) A
Fault/output/stuck_at:(1) B

In this example two passes are specified with the /NUMBER_OF_PASSES qualifier and a simulation of the faults in pass 2 is specified with the /PASS qualifier. In the order of their placement in the access list signals a and b are assigned to pass 1 and canceled, c and d are assigned to pass 2.

/PERCENT=integer

Specifies the probability that a fault is inserted on a signal in the access list. A random number generator determines the signals that have faults inserted so the location of faults is statistically random. This random number generator is reset with each USE command so that a subsequent and identical SET FAULT/PERCENT command inserts faults on the same signals.

Each signal in the network is counted only once so the use of wildcards in the access list does not increase the probability of a fault insertion on any signal.

/PHANTOM

/NOPHANTOM

Specifies whether faults are inserted on the inputs to the phantom and on the outputs of the gates driving the phantom. The default is /NOPHANTOM, which specifies that no faults be inserted.

/SCALAR

/NOSCALAR

The default, /NOSCALAR, inserts a single fault source on each memory cell in the access list. /SCALAR is used to insert fault sources on individual bits of each memory cell.

Examples

Where each cell MEM is 32 bits:

FAULT MEM[2]  ! inserts 1 fault source on MEM
    ! cell location 2
FAULT/SCALAR MEM[2]  ! inserts 32 fault sources on MEM
    ! cell location 2
When the /SCALAR qualifier is specified and the memory cell name in the access list has no bit subscript, an individual fault source is inserted on each bit in the cell. If the cell name has a bit subscript, faults are inserted only on the specified bits. The maximum number of faults that can be inserted at one cell location is 256.

Examples

Where each cell MEM is 32 bits:

- \text{FAULT/SCALAR MEM[2]} \quad \text{! inserts 32 fault sources on MEM cell location 2}
- \text{FAULT/SCALAR MEM[2]<15:0>} \quad \text{! inserts 16 fault sources on MEM cell location 2}

Note: You may not insert identical faults in the same memory cell. The faults may overlap, but they must differ by fault type or by bit location. See the Examples Section at the end of the command description for more details on this feature.

/SCOPE=scope\_name

Specifies the SCOPE setting for this FAULT command only. Differs from SET SCOPE, which specifies the scope setting for all commands.

/STUCK\_AT=(state\_list)

Inserts the faults specified in the state list, and "sticks" the specified signals at those states.

/THRESHOLD = integer

\text{/NOTHRESHOLD}

A ZYCAD–specific qualifier that specifies how many times each fault source specified with this qualifier is detected before it is dropped. If no value is specified, or if /THRESHOLD is not specified, the threshold value defaults to 1. The maximum threshold value is 65534. If more than one threshold value is specified for the same fault source, the highest value is used.

/NOTHRESHOLD specifies that each fault source be detected a maximum of 65534 times before it is dropped. /THRESHOLD=0 is equivalent to /NOTHRESHOLD.

The FAULT command inserts a source fault for each signal or element described in the command. By using the various qualifiers, you can fault single signals or elements, lists of signals or elements, or percentages of signals.

DECSIM does not insert faults on unconnected primary inputs of a network. Unconnected primary inputs are those primary inputs that don't drive fanouts.

To make simulation run as fast as possible, DECSIM collapses equivalent faults. At fault insertion time, DECSIM classifies equivalent faults into a single class and inserts only one fault from each class. For example, if the input of an AND gate is stuck at 0 and the output of the same AND gate is also stuck at 0, DECSIM classifies these two faults as equivalent and inserts just the fault on the output. Fault collapsing does not affect fault detection, however. When a fault is detected, DECSIM reports all the faults in that class as detected.
Use the FAULT command when the network is loaded and before any
simulation is performed.

DECSIM reports the number of faults inserted by the FAULT command
each time the command is executed. SHOW FAULT gives a summary of
the number and type of faults inserted by all the FAULT commands.

ZYCAD Mode

Fault simulation in ZYCAD mode is serial or concurrent, depending
on which accelerator you use; whereas in “soft DECSIM” it is always
concurrent. This means that “soft” DECSIM may actually be faster than
FAULT simulations on the LE of more than 1,000 signals. However,
the FE, which performs concurrent fault simulation, may be equal to
or faster than DECSIM. For more information on the ZYCAD hardware
accelerators, see the ZYCAD command.

The ZYCAD serial fault simulation procedure is as follows:

1  The good network is simulated.

2  ZYCAD saves the simulation results for comparison at the detection
    points you specify.

3  ZYCAD applies a fault and simulates the network again.

4  At the detection points, ZYCAD compares the simulation results with
    those from the good machine. Failure of results to match is considered
    a fault.

5  ZYCAD proceeds to step 6 when a fault is detected, or when all pattern
    stimuli are exhausted.

6  ZYCAD reinitializes the simulation environment.

7  ZYCAD loops through steps 3 through 6 until the fault list is
    exhausted.

The format and the qualifiers are the same as for “soft DECSIM”.

For more information see:

CONTEXT       EXAMINE
DETECT         ZYCAD
DROP

EXAMPLES

SIM> FAULT/INPUT/OUTPUT/STUCK AT= (0,1) E1, E2, E3
SIM> FAULT/PERCENT=50/EXCEPT=(Data_Chip) LSI_11

The following examples show the use of the /SCALAR qualifier on memory
cell fault insertion. FID is the fault identification number assigned by
DECSIM to the fault when it is defined.
USE memblock

Reading File: DOC$DISK:[SAWA.DECSIM.MEMORY]MEMBLOC\$T.NOB;12

%I-TRS, Time Resolution = 10 nS, Default unit is nS

FAULT/STUCK:(0) A1.memblock[0:2] ! FID (1),(2),(3)
FAULT/STUCK:(0) A1.memblock[0]

%E-NFI, No faults inserted

FAULT/STUCK:(1) A1.memblock[0] ! FID (4)
FAULT/STUCK:(0) A1.memblock[0]<3:0> ! FID (5)
FAULT/STUCK:(0)/SCALAR A1.memblock[0]<3:0> ! FID (6),(7),(8),(9)
FAULT/STUCK:(0)/SCALAR A1.memblock[0]<3:1>

%E-NFI, No faults inserted

FAULT/STUCK:(0)/SCALAR A1.memblock[0]<4:0> ! FID (A)

SHO FAULT/INDEX/UNDETECTED

! Total faults inserted = 10
! Fault effect limit = NO LIMIT
! Faults detected = 0 ( 0.000%)
! Firm = 0 ( 0.000%)
! Possible = 0 ( 0.000%)
! Faults removed = 0 ( 0.000%)

! Type OUTPUT INPUT MEMORY TOTAL
! -------------------------------
! S80 0 0 9 9
! S81 0 0 1 1

Undetected Faults

| Fault/stuck_at:(0) A1.MEMBLOCK[0]<7:0> ! (1) |
| Fault/stuck_at:(0) A1.MEMBLOCK[1]<7:0> ! (2) |
| Fault/stuck_at:(0) A1.MEMBLOCK[2]<7:0> ! (3) |
| Fault/stuck_at:(1) A1.MEMBLOCK[0]<7:0> ! (4) |
| Fault/stuck_at:(0) A1.MEMBLOCK[0]<3:0> ! (5) |
| Fault/scalar/stuck_at:(0) A1.MEMBLOCK[0]<0> ! (6) |
| Fault/scalar/stuck_at:(0) A1.MEMBLOCK[0]<1> ! (7) |
| Fault/scalar/stuck_at:(0) A1.MEMBLOCK[0]<2> ! (8) |
| Fault/scalar/stuck_at:(0) A1.MEMBLOCK[0]<3> ! (9) |
| Fault/scalar/stuck_at:(0) A1.MEMBLOCK[0]<4> ! (A) |
CANCEL FAULT

Removes the faults listed in the access_list and any associated fault effects from the network. You can use CANCEL FAULT prior to and during simulation.

FORMAT

CANCEL FAULT  [/qualifiers] access_list

Default Qualifiers

None.

Optional Qualifiers

/INDEX=(fid_list)
/INPUT
/OUTPUT
/STUCK_AT=(state,[state...])

restrictions

• Once faults are canceled, they cannot be reinserted.

QUALIFIERS

/INDEX = fault_ID

Removes faults that have been specified by fault ID number. The fault ID is a unique number assigned by DECSIM to each fault when it is inserted. To see the ID numbers of the faults, use SHOW FAULT/INDEX with a qualifier to specify which class of faults you wish to see (for example, /CANCELLED, /FULL, /DETECTED, etc.).

More than one fault_ID can be specified by separating them with commas and enclosing the whole argument in parentheses (for example, CANCEL FAULT/INDEX = (117#16, 11F#16, 220#16).

/INPUT

Removes faults that have been inserted on the input nodes specified in the access_list.

/OUTPUT

Removes faults that have been inserted on the output nodes specified in the access_list.

/OUTPUT is the default.

/STUCK_AT = (state,[state,...])

Specifies the types of faults to be removed from the nodes listed in the access_list, where the type of fault can be stuck at 1, stuck at 0, stuck at U, or stuck at Z. The default is (0,1)
CANCEL FAULT is used to remove fault sources and any associated fault effects from the network. Simulation proceeds as if the faults had never been inserted.

Canceled faults cannot be reinserted.

SHOW FAULT and SHOW FAULT/STATISTICS reflect the cancellations. That is, no information on the canceled faults is displayed. To see a list of canceled faults, use SHOW FAULT/CANCELLED.

See also: DROP and DETECT

EXAMPLE

SIM> CANCEL FAULT CLK, SET, RESET
SHOW FAULT

Displays characteristics of the fault simulation, such as how many faults have been inserted and how many faults have been detected.

FORMAT

SHOW FAULT  [/qualifier ... ] [machine_descriptor]

Default Qualifiers
None.

Optional Qualifiers
/CANCELLED
/DETECTED
/FIRM_DETECTS
/FULL
/INDEX
/LOGNAME=(logname [,logname...])
/POSSIBLE_DETECTS
/REMOVED
/SIZE
/STATISTICS
/UNDETECTED

ZYCAD-specific Qualifiers
/FE_FAULTS_PER_PASS
/ZYCAD

restrictions
None.

ARGUMENTS

machine_descriptor
A description of a faulty machine that contains a fault source. Machine descriptors are in curly braces {} and begin with OSA or ISA, which are abbreviations for "output stuck at" and "input stuck at", respectively. The abbreviation, OSA or ISA, is followed by a 1 or a 0 that indicates whether the output or input is stuck at 1 or 0. A colon (:) and the name of the faulty node follow. For example: {OSA1:CLK}. See Section 4.3.8 for more information on machine descriptors.

QUALIFIERS

/CANCELLED
Displays a list of the faults explicitly removed by the user with CANCEL FAULT. Canceled faults cannot be reinserted.

/DETECTED
Displays those faults that have been detected at the detection points. The display is in the format of an indirect file, and can be used to reinsert the faults.
/FE_FAULTS_PER_PASS
A ZYCAD–specific qualifier that applies to fault simulation on the ZYCAD/FE simulator, the concurrent fault evaluator. /FE_FAULTS_PER_PASS shows the upper limit on the number of faults ZYCAD inserts per concurrent simulation pass. This limit is specified with the SET FAULT/FE_FAULTS_PER_PASS command. (See SET FAULT/FE_FAULTS_PER_PASS for details.)

/FIRM_DETECTS
Lists the number of FIRM faults that have been detected and the percentage of coverage that the detections give. It also lists the names of the detected faulty signals and the type of faults that were placed on the signals: output stuck at 1 or 0, or input stuck at 1 or 0. This list is in the same format as an indirect file and can be used to reinsert the faults.

By default, the criteria for a FIRM detection is when an observation point in the good machine is 1 and the corresponding point in the faulty machine is 0 or vice versa. However, you can override this criteria by using the qualifier /FIRM with the DETECT command.

/FULL
Displays the entire list of faults currently inserted.

/INDEX
Shows the numeric indexes of faulty nodes when a list of faulty nodes is displayed with an additional qualifier such as /FULL, /POSSIBLE_DETECTS, /FIRM_DETECTS, /UNDETECTED, /DETECTED, or /CANCELLED.

/LOGNAME=logname
/LOGNAME=(logname, ...)
Routes output (from this SHOW FAULT command only) to a log file. If a logname has previously been set up with a LOG command, output is routed to this log file. Otherwise, an implicit SET LOG command is performed, which opens a log file. The log file remains open until you explicitly close it with CANCEL LOG. Multiple lognames can be used to print to multiple files by enclosing the list in parentheses.

Without the /LOGNAME qualifier, SHOW FAULT defaults to printing on the terminal.

/POSSIBLE_DETECTS
Lists the number of POSSIBLE faults that have been detected and the percentage of coverage that these detections give. It also lists the names of the detected faulty signals and the types of faults that were placed on the signals: output stuck at 1 or 0, or input stuck at 1 or 0. The display is in the same format as an indirect file and can be used to reinsert the faults.

By default, no faults are detected as possible detections. However, you can specify the criteria for possible fault detections by using the /POSSIBLE qualifier with the DETECT command.

Note: If a fault is detected both as a POSSIBLE fault and as a FIRM fault, DECSIM reports the detection as a FIRM fault detection and not as a POSSIBLE fault detection.
SHOW FAULT

/REMOVED
Displays those faults removed during simulation by the simulator by using mechanisms such as DETECT/THRESHOLD and FAULT/LIMIT.

/SIZE
Displays the number of fault effects in a faulty machine. SHOW FAULT/SIZE must be used in conjunction with another option that lists faults (e.g. /FULL, /DETECTED, /UNDETECTED, /CANCELLED), as the display is on a per fault basis. The size value is printed for output fault sources only.

/STATISTICS
Displays fault insertion and simulation statistics.

/UNDETECTED
Displays faults that have not been detected at the detection points.
If you specify /LOG with /UNDETECTED, the undetected fault display is in a form that can be used in an indirect file. If you edit out everything from the file except for the display, you can use the log file to insert previously undetected faults. An example of an edited file is shown below:

```
! Undetected Faults
!----------------------------------------
Fault/ou/st:(0) "A1"
Fault/ou/st:(1) "A1"
Fault/ou/st:(0) "A2"
Fault/ou/st:(1) "A2"
Fault/ou/st:(1) "CO"
```

/ZYCAD
Used only in ZYCAD runs. Provides additional fault reporting information in the .ZOF file. This qualifier must be used with /UNDETECTED to get a list of undetected faults in the summary ZYCAD output.

DESCRIPTION
The SHOW FAULT command displays how many:

- Stuck–at–1 and stuck–at–0 faults have been inserted on the inputs and outputs.
- FIRM and POSSIBLE faults have been detected and the percentage of coverage these detections yield.
- FIRM faults have been detected and the percentage of coverage these detections yield.
- POSSIBLE faults have been detected and the percentage of coverage these detections yield.
- Faults have been removed.

Note: Faults placed on primary inputs are shown as faults on output signals.
SHOW FAULT

When you specify a machine descriptor, DECSIM lists only the nodes on which the specified fault source has an effect. If DEBUG/STATISTICS is enabled, a summary table of the number and type of fault effects is generated.

For more information see:

DETECT       DROP

---

EXAMPLES

SIM> SHOW FAULT/LOG-log1/DETECT
SIM> SHOW FAULT/UNDETECT
| Total faults inserted = 24 |
| Faults detected = 5 ( 20.833%) |
| Firm = 5 ( 20.833%) |
| Possible = 0 ( 0.000%) |
| Faults removed = 0 ( 0.000%) |

<table>
<thead>
<tr>
<th>Type</th>
<th>OUTPUT</th>
<th>INPUT</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>S@0</td>
<td>12</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>S@1</td>
<td>12</td>
<td>0</td>
<td>12</td>
</tr>
</tbody>
</table>

Undetected Faults

Fault/ou/st:(0) "A1"
Fault/ou/st:(1) "A1"
Fault/ou/st:(0) "A2"
Fault/ou/st:(1) "A2"
Fault/ou/st:(0) "B1"
Fault/ou/st:(1) "B1"

SIM> SHOW FAULT/FULL/INDEX
| Total faults inserted = 24 |
| Faults detected = 0 ( 0.000%) |
| Firm = 0 ( 0.000%) |
| Possible = 0 ( 0.000%) |
| Faults removed = 0 ( 0.000%) |

<table>
<thead>
<tr>
<th>Type</th>
<th>OUTPUT</th>
<th>INPUT</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>S@0</td>
<td>12</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>S@1</td>
<td>12</td>
<td>0</td>
<td>12</td>
</tr>
</tbody>
</table>

All Faults Inserted

{OSA0:A1} | {1.17}
{OSA1:A1} | {1.1F}
{OSA0:A2} | {2.17}
{OSA1:A2} | {2.1F}
{OSA0:B1} | {3.17}

---

5-87
FOR

Allows you to iterate a sequence of DECSIM commands by using a control variable or index.

FORMAT

[label:] FOR loop_var FROM start_expr TO finish_expr
[BY loop_count_expr]
DO stmt

Default Qualifiers
None.

Optional Qualifiers
None.

restrictions

- You may not abbreviate the keywords: FOR, FROM, TO.
- FROM and TO are required.
- BY = 1 or -1.

DESCRIPTION

FOR operates as follows:

1 DECSIM evaluates, and saves as loop limit constants, the "FROM start" and "TO finish" expressions.

2 DECSIM initializes the loop_var (which you have defined) to the start loop limit.

3 DECSIM compares the loop_var with the TO finish limit.

4 If the current value of the loop_var is less than or equal to the TO finish value, DECSIM executes the DO statement. (If descending order is used, a DO statement will be executed if the current value of the loop_var is greater than or equal to the TO finish value). A statement containing the loop_var uses the current value of the variable.

5 After executing the DO statement, DECSIM adds the BY value to the loop_var, and returns to step 3.

If the BY expression evaluates to 0, it will be changed to 1 or -1, depending on the values of FROM and TO. No warning is issued in this case.

The loop_var must be a simple name (that is, no spaces; underscore is permitted, however) with no bit subscripts (in the FOR header) and no word indices. If the loop_variable has been defined in a STATE command, the new loop_variable does not alter the value of the old variable. However, the old value is not accessible from within the loop.

DECSIM does not check on whether an EVALUATE command in the loop explicitly alters the loop_variable.
If two nested loops declare the same loop_variable, the correct number of iterations will be made, but the innermost variable value will be used.

See the Examples Section in the MACRO command description for examples of how macros behave when nested within a FOR loop.

Note: In the Behavior Description Language there are also FOR loops. In that language, however, the keyword DOWNTO must be used to show descending order. See Section 2.6.3 for more information.

For more information see:

IF UNTIL
LEAVE WHILE

EXAMPLES

SIM> FOR I FROM 0 TO 15 DO
MORE> BEGIN
MORE> DEF REG_1<15:0> = 0
MORE> DEF REG_1<15> = 1
MORE> SIM
MORE> END

SIM> FOR I FROM 10 TO 1 DO PRINT I, '**2 = ',(I*I)

Control Flow Tracing

The following tracing is shown when the "DEBUG /LOOP" command is in effect.

SIM> FOR I FROM 1 TO 5 DO PRINT I,'**2 = ',(I*I)
[ FOR-loop: I = 1 ]
1**2 = 1
[ FOR-loop: I = 2 ]
2**2 = 4
[ FOR-loop: I = 3 ]
3**2 = 9
[ FOR-loop: I = 4 ]
4**2 = 16
[ FOR-loop: I = 5 ]
5**2 = 25
[ End of FOR-loop: I ]
SIM>
HALT

Returns control to command mode from an indirect file or a WATCH action.

FORMAT

HALT

Default Qualifiers
None.

Optional Qualifiers
None.

restrictions

- The HALT command in a WATCH action does not prevent succeeding commands in that WATCH action from being executed.

DESCRIPTION

HALT causes an immediate simulation halt and a return to command mode. Therefore, it can be useful in a WATCH action or indirect file, or it can be used conditionally in a loop or if statement.

For more information see:

BREAK       LEAVE ALL
DISABLE     PAUSE
INDIRECT
HAZARD

Enables you to detect time and data violations on scalar signals in equation, memory and MOS models. The types of hazard checkers you can set are: pulsewidth, cyclewidth, setup, and hold. The pulsewidth and cyclewidth checkers can perform either a level check or an edge check. The Description Section below lists the HAZARD command qualifiers that are required in each checker type. For a full description of these checker types and of the role of hazard analysis in simulation, see Section 4.5.3.

With the HAZARD command, you can define any hazard checker as a conditional hazard by specifying an enable signal. Also, you can customize the amount of information displayed when a hazard violation occurs or when a hazard check is completed successfully, and you can route that information to various log files.

If you prefer, you can define hazard checkers in your model. Section 1.4 describes how to define checkers in your model.

CANCEL, DISABLE, ENABLE HAZARD and SHOW HAZARD are described following the [SET] HAZARD command description. See also [SET] LOG/HAZARD.

FORMAT

[SET] HAZARD /TYPE=hazard_type [/qualifier...] access_list

Default Qualifiers

/EDGE=REFERENCE=X_X I for setup, hold checkers only
/NOENABLE I for all checker types
/NOLOGNAME I for all checker types
/NOMAX_WIDTH I for pulse, cyclewidth checkers only
/NOMIN_WIDTH I for pulse, cyclewidth checkers only
/NOOFFSET I for setup, hold checkers only
/REPORT=(DATA_VIOLATION, FULL, NOFIRST, NOSUCCESS, TIME_VIOLATION) I for all checker types
/STATE=DATA=X I for pulse, cyclewidth checkers only
/STATE=DATA=1 I for conditional checkers only
/STATE=ENABLE=1

Required Qualifiers

/ENABLE=signal_name I for conditional checkers only
/TYPE=hazard_type I for all checkers

/MAX_WIDTH=time I for pulsewidth, and/or MIN_WIDTH=time I cyclewidth checkers only

/REFERENCE=signal_name I for setup, hold checkers only
/WINDOW=time I for setup, hold checkers only
Optional Qualifiers

/EDGE=[NO]DATA=edge      ! for pulse, cyclewidth checkers only
/EDGE=[NO]REFERENCE      ! for setup, hold checkers only
         =edge_list
/HAZARDNAME=hazard_name  ! for all checker types
/LOGNAME=logname_list    ! for all checker types
/OFFSET=DATA=time        ! for setup, hold checkers only
/NOEDGE                   ! for all checker types
/NOREFERENCE              ! for setup, hold checkers only
/NOSTATE                  ! for all checker types
/NOWINDOW                 ! for setup, hold checkers only
/[NO]REPORT=(BRIEF,
           ALL,
           NODATA_VIOLATION,
           FIRST,
           NONE,
           SUCCESS,
           NOTIME_VIOLATION
/)STATE=[NO]DATA=state    ! for all checker types
/)STATE=[NO]DATA=state_list ! for setup, hold checkers only
/)STATE=[NO]ENABLE=state_list ! for conditional checkers only

restrictions

- Hazard checkers are not allowed on vector outputs (behavioral or memory).
- Hazard checkers are ignored in ZYCAD networks and during fault simulation.

QUALIFIER FIELDS

The qualifier fields that are unique to SET HAZARD are defined below.

state
One of the four DECSIM states (0, 1, U, Z). The #2 specification is not allowed. Wildcards are not allowed.

design
A single transition expressed as state_state, where state is one of the four DECSIM states. The #2 specification is not allowed. Wildcards are not allowed.

state_list
A single state or a list of states separated by commas and enclosed in parentheses. The wildcard X may be used. Thus /STATE=DATA=X means the same as /STATE=DATA=(0,1,U,Z).

design_list
A single edge or a list of edges separated by commas and enclosed in parentheses. The wildcard X may be used to represent one or both states in a transition. Thus /EDGE=REFERENCE=X_1 is equivalent to /EDGE=REFERENCE=(U_1,Z_1,0_1), and /EDGE=REFERENCE=X_X means any transition on the reference signal.
time
A single time value, with or without time units. The default time unit is nanoseconds.

QUALIFIERS

/EDGE=DATA=edge
/EDGE=NODATA
/EDGE=DATA can be used only with pulsewidth or cyclewidth checkers to specify that when the data signal (usually a repeating signal such as a clock) makes the specified transition, the waveform must be square. For example, if you specify a 0→1 edge, 0→1→0 is a square pulse, and 0→1→0 is a square cycle; 0→1→U#2 is not square. There is no default edge specification. You cannot use this qualifier in conjunction with the /STATE=DATA qualifier.

/EDGE=NODATA cancels a specified edge on the data signal.

Example
/EDGE=DATA=0_1
/EDGE=DATA=U_0

/EDGE=REFERENCE=edge_list
/EDGE=NOREFERENCE
/EDGE=REFERENCE can be used only with setup and hold checkers to specify the edge on the reference signal that marks the end or beginning of the hazard window, respectively. If this qualifier is not specified, the default is any transition on the reference signal (X_X). Use /REFERENCE=signal_name to identify the reference signal.

/EDGE=NOREFERENCE cancels a user-specified reference edge in a setup or hold checker and returns to the default, (X_X).

Examples
/EDGE=REFERENCE=0_1
/EDGE=REFERENCE=X_1
/EDGE=REFERENCE=(0_1, 1_0)

/NOEDGE
Cancels user-specified edges on both the data and reference signals.

/ENABLE=signal_name
/NOENABLE
/ENABLE can be used with any type of hazard checker to identify a single enable signal. (The wildcard * is not allowed.)

When /ENABLE is used, the checker is processed only if the enable signal is at a particular state or set of states when the checker is activated. (See Section 4.5.3.4 for more information on these conditional checkers.) If no state is specified with the /STATE=ENABLE=state_list qualifier, the default state of the enable signal is 1.

/NOENABLE is the default and also can be used to cancel a user-specified enable signal.

Example
/ENABLE=d1.c2.b1
HAZARD

/HAZARDNAME=hazard_name
Can be used to specify a name for the hazard checker, which is then used to reference the checker in SHOW, CANCEL, ENABLE, DISABLE HAZARD commands. If this qualifier is not used in the hazard checker definition, DECSIM generates a default name whose format is HAZARDnumber.

If the hazard checker includes an access list of more than one signal, the hazard name, whether assigned by DECSIM or by the user, applies to all the hazard checkers set on the signals in the access list. Despite being defined as a group, each valid primary signal listed in the access list is defined and processed as a separate hazard checker with a common hazard name and a common set of parameters.

Example

/HAZARDNAME=cyclechecker

/LONGNAME=longname_list

/NOLOGNAME
/LONGNAME can be used with any hazard checker to specify a log file or files to receive the information generated by the hazard checker. If a log file with that name has already been opened by a previous hazard check definition, the information is simply routed to that file. If a log file with that name does not exist, it is opened with the /NORESPONSES, /NOCOMMANDS, and /NOECHO qualifiers when the hazard checker is defined. The default extension is .HAZ.

/NOLOGNAME is the default and routes hazard information to the screen only.

Example

/LONGNAME=(cyclechecker, clockhazards)

Note: This qualifier cannot be used to direct hazard information from NETPRO hazard checkers (checkers defined in the structural model) to a log file. To capture hazard information from NETPRO checkers, you must open a log file with the [SET] LOG/RESPONSES command before compiling or loading a model. You may use the /HAZARD qualifier on the LOG command to put a .HAZ extension on the log file.

/MAX_WIDTH=time

/NOMAX_WIDTH
/MAX_WIDTH can be used only with pulselidth and cyclewidth checkers to define the maximum time window during which the pulse or cycle duration is checked. This qualifier must be used, if the /MIN_WIDTH qualifier is not used; both /MIN_WIDTH and /MAX_WIDTH qualifiers may be used together.

/NOMAX_WIDTH is the default and can also be used to cancel a user-specified maximum time window.

Example

/MAX_WIDTH=1000
/MIN_WIDTH=time
/NOMIN_WIDTH

/MIN_WIDTH can only be used with pulsewidth and cyclewidth checkers to define the minimum time window during which the pulse or cycle duration is checked. This qualifier must be used if the /MAX_WIDTH qualifier is not used; both /MIN_WIDTH and /MAX_WIDTH qualifiers may be used together.

/NOMIN_WIDTH is the default and can also be used to cancel a user-specified minimum time window.

Example

/MIN_WIDTH=90

/OFFSET=DATA=time
/NOOFFSET

/OFFSET=DATA can be used only with setup or hold checkers to offset the hazard window relative to the reference edge. In a setup checker, this qualifier ends the setup window offset time units before the reference edge. In a hold checker, this qualifier starts the hold window offset time units after the reference edge. A transition that occurs during the offset is not reported as a violation.

/NOOFFSET is the default and can also be used to cancel a user-specified offset.

Example

/OFFSET=DATA=10 ps

/REFERENCE=signal_name
/NOREFERENCE

/REFERENCE must be used in setup and hold checkers to identify the single signal whose transition marks the end of the setup window or the beginning of the hold window, respectively. The /EDGE=REFERENCE qualifier specifies a particular edge or set of edges of the reference signal; otherwise, the data signal is checked when any transition in the reference signal occurs. The wildcard * is not allowed in the signal name.

/NOREFERENCE can be used to cancel a user-specified reference signal.

Example

/REFERENCE=d1.clock

/REPORT=keyword
/REPORT=(keyword, keyword [,keyword...])
/NOREPORT

/REPORT can be used to customize the amount of information reported by the checker. The exact format of the report depends on the combination of the /REPORT keywords, the result of the hazard check, the hazard type, and the specified hazard parameters.

/NOREPORT is equivalent to /REPORT=NONE and turns off hazard reporting for the checker.
The default REPORT format is:

DATA_VIOLATION
FULL
NOSTART
NOSUCCESS
TIME_VIOLATION

All hazard results will be reported with a message code that identifies the following information:

1. Message code: %HAZ
2. Hazard type: PUL, CYC, SET, HOL (first three letters of type)
3. Checker result: SU, DV, TV (success, data, or time violation)

The following sections describe the kinds of information that will be reported with each of the keywords.

ALL
Is equivalent to TIME_VIOLATION, DATA_VIOLATION, SUCCESS and specifies that all hazard checker successes and violations, including both data and time, be reported for the associated hazard checker.

BRIEF
Specifies that a limited amount of information be printed in the event of a hazard violation, including the message code, the simulation time, the name of the hazard checker, and the signal(s).

Example

SIM> SET HAZARD /TYPE=cycle /EDGE=data=0_1 /MAX=25 -
/REP=(brief,suc) strobe2
%HGN, HAZARD name is HAZARD2

%HGN CYC TV | 195 nS HAZARD2 STROBE1

[NO]DATA_VIOLATION

DATA_VIOLATION is the default and specifies that hazard checker data violations are reported. The NODATA_VIOLATION keyword turns off all reporting of data violations associated with the specified hazard checker.

[NO]FIRST

NOFIRST is the default and indicates that all time violations, not just the first one, are reported for the specified hazard checker. FIRST limits the reporting of hazard time violations to the first time violation of a given hazard only. It does not apply to the reporting of hazard checker successes and data violations.
FULL

Specifies that information about the nature of the hazard violation or success be printed in addition to the general output that appears with the BRIEF format. The FULL format includes the expected and actual timing of the transition and the simulation time when the report was made. In setup and hold checkers, the name of the reference signal is also included.

The keywords FULL and BRIEF are mutually exclusive. The default is FULL.

Example

```
SIM> SET HAZ/TYPE=pulsewidth/min_width=10 -
    /STATE=data=1/REPORT=(success,full) D1
%HGN, HAZARD name is HAZARD1
.

%HAZ PUL SU |31 nS   HAZARD1
Exp: 10 nS   Act: 11 nS
Simulation Clock: |40 nS
```

NONE

Is equivalent to NODATA_VIOLATION, NOTIME_VIOLATION, NOSUCCESS and turns off all hazard reporting features for the specified hazard checker. NONE is the opposite of ALL.

[NO]SUCCESS

The default is NOSUCCESS, which turns off all reporting of successes associated with the specified hazard checker. SUCCESS specifies that hazard checker successes be reported.

[NO]TIME_VIOLATION

TIME_VIOLATION is the default and specifies that hazard checker time violations be reported. NOTIME_VIOLATION turns off all reporting of time violations associated with the specified hazard checker.

/STATE=DATA=state

/STATE=DATA=state_list

/STATE=NODATA

/STATE=DATA=state can be used with pulsewidth or cyclewidth checkers to check the duration of the wave form when it is at the specified state. The transition edges to and from the data state are not checked.

You cannot use this qualifier in pulsewidth or cyclewidth checkers in conjunction with the /EDGE=DATA qualifier. If neither qualifier is specified in a pulsewidth or cyclewidth checker, /STATE=DATA=1 is the default.

/STATE=DATA=state_list can be used in setup and hold checkers to check whether the signal is at the specified state or not during the setup or hold window. If this qualifier is not used, the default is /STATE=DATA=X.

/STATE=NODATA cancels the specified state or state_list and returns to the default for that checker type.
HAZARD

Example

/STATE=DATA=U
/STATE=DATA= (0, 1)

/STATE=ENABLE=state_list
/STATE=NOENABLE

/STATE=ENABLE can be used with any hazard checker to specify the state of the enable signal, which must be identified with the /ENABLE qualifier. Together with the /ENABLE qualifier, the /STATE=ENABLE qualifier defines a conditional hazard, a hazard that is monitored only when the enable signal is at a particular state or set of states. If the /STATE=ENABLE qualifier is not used, the default state of the enable signal is 1. For more information on conditional checkers, see Section 4.5.3.4.

/STATE=NOENABLE cancels the user-specified state of the enable signal and returns to the default.

Example

/STATE=ENABLE= (1, 2)

/NOSTATE

Cancels user-specified edges on both the data and enable signals.

/TYPEn= hazard_type

Must be used with all command language hazard checkers to determine the type of check being defined. You must use one of the four mutually exclusive keywords to specify the type: PULSEWIDTH, CYCLEWIDTH, SETUP, or HOLD. For more information on the four types of checkers, see Section 4.5.3.4.

Example

/TYPEn=PULSEWIDTH

/WINDOW=time

/NOWINDOW

/WINDOW must be used with setup and hold checkers only to define the hazard window. In a setup checker, the /WINDOW qualifier specifies the minimum time the data signal must be stable before the edge on the reference signal. In the hold checker, the /WINDOW qualifier specifies the minimum time the data signal must be stable beginning with the edge on the reference signal. The /OFFSET qualifier can be used to offset the window in either checker type.

/NOWINDOW cancels a user-specified window.

Example

/WINDOW=15
DESCRIPTION

To set a hazard checker from the command language, you must choose the type of hazard you want to be reported and the signals you want to apply it to. The following sections describe the four types of hazard checkers and the qualifiers that are specific to them. The last section describes conditional checkers. For a detailed discussion of the boundary conditions of these checkers and the violations they report, see Section 4.5.3.4.

Pulsewidth and Cyclewidth Checkers

Pulsewidth and cyclewidth checkers check the waveform of a repeating signal, such as a clock, during a window of time specified by the user. A pulsewidth is defined as a transition to and from a specified state. A cyclewidth is a transition to a specified state, a transition to another state or states, and a transition back to the specified state.

Table 5–1 shows the qualifiers that define pulsewidth and cyclewidth checkers.

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Use</th>
<th>Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td>/TYPE=pulsewidth</td>
<td>required</td>
<td>none</td>
</tr>
<tr>
<td>/MAX_WIDTH=time</td>
<td>required if no MIN_WIDTH</td>
<td>none</td>
</tr>
<tr>
<td>/MIN_WIDTH=time</td>
<td>required if no MAX_WIDTH</td>
<td>none</td>
</tr>
<tr>
<td>/STATE=DATA=state</td>
<td>optional, can’t use with /EDGE=DATA</td>
<td>state=1</td>
</tr>
<tr>
<td>/EDGE=DATA=edge</td>
<td>optional, can’t use with /STATE=DATA</td>
<td>none</td>
</tr>
</tbody>
</table>

Setup and Hold Checkers

A setup checker defines a minimum time window during which the data signal must be stable before an edge on the reference signal. A hold checker is symmetrically opposite to a setup checker, relative to the reference edge. The hold checker defines a minimum time window during which the data signal must be stable beginning with an edge on the reference signal.

Table 5–2 shows the qualifiers that define setup and hold checkers.

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Use</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>/TYPE=setup</td>
<td>required</td>
<td>none</td>
</tr>
<tr>
<td>/WINDOW=time</td>
<td>required</td>
<td>none</td>
</tr>
<tr>
<td>/REFERENCE=signal_name</td>
<td>required</td>
<td>none</td>
</tr>
<tr>
<td>/EDGE=REFERENCE=edge_list</td>
<td>optional</td>
<td>edge=X_X</td>
</tr>
<tr>
<td>/OFFSET=DATA=time</td>
<td>optional</td>
<td>none</td>
</tr>
<tr>
<td>/STATE=DATA=state_list</td>
<td>optional</td>
<td>X</td>
</tr>
</tbody>
</table>
Table 5-3  Conditional Checker Qualifiers

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Use</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ENABLE=signal_name</td>
<td>required</td>
<td>none</td>
</tr>
<tr>
<td>/STATE=ENABLE=state_list</td>
<td>optional</td>
<td>1</td>
</tr>
</tbody>
</table>

EXAMPLES

SIM> HAZARD/TYPE=PULSE/MIN=10/MAX=20/REP=ALL strobe0
%I-HGN, HAZARD name is HAZARD1
SIM> DIF strobe0=[0, 1:10, 0:10, 1:10, 0:5, 1:5, 0:30, 1:10, U#2:10, 0:10]
SIM> SIM 100
%HAZ PUL SU |20 nS  HAZARD1  STROBE0
            Exp: 10 nS  Act: 10 nS
%HAZ PUL TV |35 nS  HAZARD1  STROBE0
            Exp: 10 nS  Act: 5 nS
%HAZ PUL TV |60 nS  HAZARD1  STROBE0
            Exp: 20 nS  Act: 20 nS
            Maximum violation, checker timed out
%HAZ PUL SU |90 nS  HAZARD1  STROBE0
            Exp: 10 nS  Act: 10 nS

Simulation Clock: |100 nS

SIM> HAZARD/TYPE=CYCLE/MAX=25/EDGE=DATA=0_1/REP=(ALL,BRIEF) strobe0
%I-HGN, HAZARD name is HAZARD2
SIM> DIF strobe0=[0, 1:10, 0:10, 1:10, U#2:5, 0:5, 1:20, 0:20, 1:10]
SIM> SIM 100
%HAZ CYC SU |30 nS  HAZARD2  STROBE0
%HAZ CYC DV |35 nS  HAZARD2  STROBE0
%HAZ CYC TV |85 nS  HAZARD2  STROBE0

Network idle, Simulation Clock: |100 nS

SIM> HAZARD/TYPE=SETUP/MIN=10/EDGE=REF=(0,1)_MORE=/REF=strobe0/REF=(ALL,BRIEF) d1
%I-HGN, HAZARD name is HAZARD3
SIM> DIF d1=[0, 1:10, 0:10, U#2:10, 0:10, 1:15, 0:5, 1:12, 0:0]
SIM> DIF strobe0=[0, [1:10, 0:10] INF]
SIM> SIM 100
%HAZ SET SU |20 nS  HAZARD3  D1
%HAZ SET DV |40 nS  HAZARD3  D1
%HAZ SET TV |60 nS  HAZARD3  D1
%HAZ SET TV |80 nS  HAZARD3  D1
%HAZ SET SU |100 nS  HAZARD3  D1

Simulation Clock: |100 nS

5-100
SIM> SET TIME 0

Simulation Clock: 10 ns

SIM> HAZARD/TYP=HOLD/WIN=10000PS/REF=strobe0/STATE=DATA=(0,1)-

MORE> /EDGE=REF=(0_1,0_1)/REF=ALL d0

%I-HGN, HAZARD name is HAZARD4

SIM> DEF strobe0=[U$2, [1:10,0:10,1:10, U$2:10] INF]

SIM> DEF d0=[0,1:10,0:15,1:5,0:3,1:17,0:4,1:16,0:9, U$2:11]

SIM> SIM 100

%HAZ HOL SU | 19 nS     HAZARD4    DO
Exp: 10 nS   Act: 9 nS Ref: STROBE0

%HAZ HOL TV | 33 nS     HAZARD4    DO
Exp: 10 nS   Act: 3 nS Ref: STROBE0

%HAZ HOL TV | 54 nS     HAZARD4    DO
Exp: 10 nS   Act: 4 nS Ref: STROBE0

%HAZ HOL DV | 79 nS     HAZARD4    DO
Exp: 10 nS   Act: 9 nS Ref: STROBE0

Simulation Clock: 100 ns

SIM> HAZARD/TYP=HOLD/WIN=10/REF=strobe0/EDGE=REF=1_0-

MORE> /STATE=ENABLE=1/ENABLE=strobe1/REF=(ALL,BRIEF) d0

%I-HGN, HAZARD name is HAZARD5

SIM> DEF strobe0=[0, [1:10,0:10] INF]

SIM> DEF strobe1=[0, [1:15,0:10,1:10,0:10]

SIM> DEF d0=[1,0:10, [1:20, U$2:15,0:3] INF]

SIM> SIM 100

%HAZ HOL SU | 29 nS     HAZARD5    DO

%HAZ HOL TV | 45 nS     HAZARD5    DO

Simulation Clock: 100 ns
CANCEL, DISABLE, ENABLE HAZARD

The CANCEL command enables you to remove a hazard, a group of hazards, or all hazards from a simulation session permanently. The DISABLE HAZARD command inactivates a specified hazard or hazards temporarily, and ENABLE HAZARD reactivates hazards that have been disabled.

**FORMAT**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANCEL HAZARD</td>
<td>[/qualifier] [hazard_name_list]</td>
</tr>
<tr>
<td>DISABLE HAZARD</td>
<td>[/qualifier] [hazard_name_list]</td>
</tr>
<tr>
<td>ENABLE HAZARD</td>
<td>[/qualifier] [hazard_name_list]</td>
</tr>
</tbody>
</table>

Default Qualifiers

None

Optional Qualifiers

/ALL
/BY_TYPE=hazard_type_list

**restrictions**

- One of the qualifiers or a hazard_name_list must be used in these commands.

**ARGUMENTS**

hazard_name_list

A single hazard name or list of hazard names, separated by commas. If the hazard checker is a NETPRO checker, the hazard name, whether it is user-specified or assigned by DECSIM, must be prefixed by a sequence of labels that identify its location in the model hierarchy. To reference command language checkers, use the name you specified in the hazard checker definition or the name DECSIM generates if no user_specified name is given. Use SHOW HAZARD to display the hazard names in your model.

**QUALIFIERS**

/ALL

Refers to all the defined hazards in the simulation.
/BY_TYPE=hazard_type_list
Identifies a group of hazards by type. The hazard_type_list can include a single keyword or a list of keywords enclosed in parentheses and separated by commas. The possible keywords are the following:

(NO)PULSEWIDTH  (NO)CYCLEWIDTH  (NO)SETUP  (NO)HOLD  NONE  ALL

DESCRIPTION
The CANCEL HAZARD command removes the specified hazard definitions from the simulation session. To restore NETPRO-defined hazards that have been canceled, you must reload the network. To restore command language checkers, redefine them with the [SET] HAZARD command.

The DISABLE HAZARD command removes the specified hazard checkers from the list of checkers being monitored, but does not remove them from the simulation session. The checkers are considered idle until they are enabled or canceled.

The ENABLE HAZARD command reverses the action of the DISABLE HAZARD command by reactivating the specified checkers.

EXAMPLES

SIM> SHOW HAZARD/ALL
LISTING of ALL HAZARDS

<table>
<thead>
<tr>
<th>Haz Enable/</th>
<th>Success</th>
<th>Data</th>
<th>Time</th>
<th>Primary</th>
<th>Hazardname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Disable</td>
<td>Vio.</td>
<td>Vio.</td>
<td></td>
<td>Signal</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>------</td>
<td>------</td>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>Hold</td>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>D0</td>
</tr>
<tr>
<td>Hold</td>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>D0</td>
</tr>
<tr>
<td>Setup</td>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>D1</td>
</tr>
<tr>
<td>Cycle</td>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>STROBE0</td>
</tr>
<tr>
<td>Pulse</td>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>STROBE0</td>
</tr>
</tbody>
</table>

HAZARD5
HAZARD4
HAZARD3
HAZARD2
HAZARD1

SIM> DISABLE HAZARD hazard1
SIM> SHOW HAZARD/ALL
LISTING of ALL HAZARDS

<table>
<thead>
<tr>
<th>Haz Enable/</th>
<th>Success</th>
<th>Data</th>
<th>Time</th>
<th>Primary</th>
<th>Hazardname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Disable</td>
<td>Vio.</td>
<td>Vio.</td>
<td></td>
<td>Signal</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>------</td>
<td>------</td>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>Hold</td>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>D0</td>
</tr>
<tr>
<td>Hold</td>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>D0</td>
</tr>
<tr>
<td>Setup</td>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>D1</td>
</tr>
<tr>
<td>Cycle</td>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>STROBE0</td>
</tr>
<tr>
<td>Pulse</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>STROBE0</td>
</tr>
</tbody>
</table>

HAZARD5
HAZARD4
HAZARD3
HAZARD2
HAZARD1

SIM> DISABLE HAZARD/BY_TYPE=HOLD
SIM> SHOW HAZARD/BY_TYPE=HOLD

5-103
# CANCEL, DISABLE, ENABLE HAZARD

---

## LISTING ---BY TYPE

<table>
<thead>
<tr>
<th>Haz Enable/Success</th>
<th>Data</th>
<th>Time</th>
<th>Primary</th>
<th>Hazardname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Disable</td>
<td>Vio.</td>
<td>Vio.</td>
<td>Signal</td>
<td></td>
</tr>
</tbody>
</table>

| Hold | D   | 0 0 0 0 D0 | HAZARD5 |
| Hold | D   | 0 0 0 0 D0 | HAZARD4 |

**SIM> ENABLE HAZARD/TYPE=HOLD**

**SIM> SHOW HAZARD/TYPE=HOLD**

---

## LISTING ---BY TYPE

<table>
<thead>
<tr>
<th>Haz Enable/Success</th>
<th>Data</th>
<th>Time</th>
<th>Primary</th>
<th>Hazardname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Disable</td>
<td>Vio.</td>
<td>Vio.</td>
<td>Signal</td>
<td></td>
</tr>
</tbody>
</table>

| Hold | E   | 0 0 0 0 D0 | HAZARD5 |
| Hold | E   | 0 0 0 0 D0 | HAZARD4 |

**SIM> CANCEL HAZARD hazard1, hazard2**

**SIM> SHOW HAZARD/ALL**

---

## LISTING of ALL HAZARDS

<table>
<thead>
<tr>
<th>Haz Enable/Success</th>
<th>Data</th>
<th>Time</th>
<th>Primary</th>
<th>Hazardname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Disable</td>
<td>Vio.</td>
<td>Vio.</td>
<td>Signal</td>
<td></td>
</tr>
</tbody>
</table>

| Hold | E   | 0 0 0 0 D0 | HAZARD5 |
| Hold | E   | 0 0 0 0 D0 | HAZARD4 |
| Setup | E | 0 0 0 0 D1 | HAZARD3 |

**SIM> CANCEL HAZARD/ALL**

**SIM> SHOW HAZARD/ALL**

%HNI, NO Hazard Checkers Inserted

---

5-104
SHOW HAZARD

Displays information about a specific hazard or group of hazards. This includes the hazardname(s), whether it is enabled or disabled, and the success and violation counts. The hazard definition includes all the signals and parameters that you specified when you defined the checker.

FORMAT

SHOW HAZARD [/QUALIFIER] hazard_name_list
SHOW HAZARD /ALL [/QUALIFIER]
SHOW HAZARD /BY_TYPE[/QUALIFIER]

Default Qualifiers
/BRIEF
/NOLOGNAME
/NOSUMMARY

Optional Qualifiers
/ALL
/BY_TYPE=hazard_type_list
/FULL
/LOGNAME=logname_list
/SUMMARY

restrictions

- /ALL or /BY_TYPE or a hazard_name_list must be used in these commands.

ARGUMENTS

hazard_name_list
A single hazard name or list of hazard names, separated by commas. If the hazard checker is defined in your model, the hazard name (whether it is user-specified or assigned by DECSIM) must be prefixed by a sequence of labels that identify its location in the model hierarchy. To reference checkers defined in the command language, use the name you specified in the hazard checker definition or the name DECSIM generates if no user-specified name is given.

QUALIFIERS

/ALL
Refers to all defined hazards in a simulation.

/BRIEF
Is the default printing format for hazard checkers. It prints only the counts and the basic definition of the checker without reporting features, states, time fields and any signals. (See Examples section below.)
/BY_TYPE=hazard_type_list
Displays information about a particular hazard type or types. The hazard_type_list can include a single keyword or a list of keywords enclosed in parentheses and separated by commas. The possible keywords are the following:

(NO)PULSEWIDTH
(NO)CYCLEWIDTH
(NO)SETUP
(NO)HOLD
NONE
ALL

/FULL
Displays all the data specified when the hazard checker was defined and includes success and violation counts. (See Examples section below.)

/(NO)LOGNAME=logname_list
/NOLOGNAME is the default and routes SHOW HAZARD output to the screen only. LOGNAME specifies log file for hazard information. The default extension is .LOG.

/(NO)SUMMARY
/SUMMARY displays the total number of successes, time violations, data violations and hazards in the simulation by hazard type. It gives the run-time totals such as the number of hazard checkers canceled, enabled, and disabled, and the number of control events scheduled and executed. NOSUMMARY is the default.

DESCRIPTION
The SHOW HAZARD command enables you to display a limited or complete amount of information about a single hazard, a group of hazards, all the hazards of a particular type, or all the hazards in a simulation. You can also display a summary of hazard information about a simulation.
EXAMPLES

SHOW HAZARD/summary/all

<table>
<thead>
<tr>
<th>Hazard Type</th>
<th>Total</th>
<th>Successes</th>
<th>Data Vio.</th>
<th>Time Vio.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pulsewidth</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cyclewidth</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hold</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Setup</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Totals</td>
<td>5</td>
<td>15</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Total control events executed = 13
Total control events scheduled = 24
Total of hazards changed = 49
Total canceled checkers = 0
Total Enabled Checkers = 5
Total Disabled Checkers = 0

LISTING of ALL HAZARDS

<table>
<thead>
<tr>
<th>Haz Enable/ Success</th>
<th>Data Vio.</th>
<th>Time Vio.</th>
<th>Primary</th>
<th>Hazardname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Disable Vio. Vio. Signal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hold E</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>D0</td>
</tr>
<tr>
<td>Hold E</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>D0</td>
</tr>
<tr>
<td>Setup E</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>D1</td>
</tr>
<tr>
<td>Cycle E</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>STROBE0</td>
</tr>
<tr>
<td>Pulse E</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>STROBE0</td>
</tr>
</tbody>
</table>

SHOW HAZARD/summary/logname=SHOWHAZ/ALL
%I-L10, Log file DOC$DISK:[SAWA.DECPSIM.HAZARDS]SHOWHAZ.LOG;4 implicitly opened

SHOW HAZARD/full hazard1,hazard2

LISTING BY HAZARD NAME

<table>
<thead>
<tr>
<th>Haz Enable/ Success</th>
<th>Data Vio.</th>
<th>Time Vio.</th>
<th>Primary</th>
<th>Hazardname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Disable Vio. Vio. Signal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pulse E</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>STROBE0</td>
</tr>
<tr>
<td>/REPORT=NoFirst</td>
<td>Time_vio</td>
<td>Success</td>
<td>Data_vio</td>
<td>Full</td>
</tr>
<tr>
<td>/STATE= 1</td>
<td>NoEnable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/MIN_WIDTH= 10 nS</td>
<td></td>
<td></td>
<td></td>
<td>Max_Width= 20 nS</td>
</tr>
<tr>
<td>Cycle E</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>STROBE0</td>
</tr>
<tr>
<td>/REPORT=NoFirst</td>
<td>Time_vio</td>
<td>Success</td>
<td>Data_vio</td>
<td>Brief</td>
</tr>
<tr>
<td>/EDGE= 0.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/MAX_WIDTH= 25 nS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SHOW HAZARD/by_type=pulsewidth

LISTING ---BY TYPE

<table>
<thead>
<tr>
<th>Haz Enable/ Success</th>
<th>Data Vio.</th>
<th>Time Vio.</th>
<th>Primary</th>
<th>Hazardname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Disable Vio. Vio. Signal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pulse E</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>STROBE0</td>
</tr>
</tbody>
</table>
HELP

Gives simple on-line assistance in using the DECSIM command language. HELP prints brief messages to jog your memory; the DECSIM manuals provide more complete information.

FORMAT

[SHOW] HELP
[SHOW] HELP  keyword_or_topic [sub_topic
[sub_sub_topic ...]]

Default Qualifiers
None.

Optional Qualifiers
None.

restrictions
None.

DESCRIPTION

HELP consists of DECSIM keywords or topics, many of which are divided into subtopics.

When you type HELP, DECSIM prints a short explanation of the command. It then lists keywords and topics for which help is available, and guides you through the layers of information by using prompts.

For quick help on syntax problems, you may want to use escape prompting. For more information see:

   Escape prompting (see Section 4.4.2).

EXAMPLES

SIM> HELP
HELP Command

The HELP command supplies specific on-line information for DECSIM commands, concepts, and features. The forms of the HELP command are:

SIM> HELP
Displays the top-level choices.

SIM> HELP topic

Types specific information about the feature(s) requested and gives next-level choices.

5–108
Note that "topic" can be more than one keyword/topic to see help for a specific (hierarchically nested) subtopic. Each level of topic can be abbreviated. HELP itself can be abbreviated to "H".

Examples: HELP RADIX
HELP SET LOG /RESPONSES !Be sure to include "/"
HELP PRINT Arguments
HELP EXPRESSION OPERATOR AND

For more information about the DECSIM HELP command, type HELP HELP. Information is available in HELP for the following:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Access_lists</th>
<th>ACTIVATE</th>
<th>ALLOCATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN-END</td>
<td>CALL</td>
<td>COMMER</td>
<td>COMMENT</td>
</tr>
<tr>
<td>BREAK</td>
<td>CANCEL</td>
<td>Commands</td>
<td>COMMENT</td>
</tr>
<tr>
<td>Comments</td>
<td>CONFIGURE</td>
<td>Continuations</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>Control_Flow</td>
<td>CTRL/C</td>
<td>CTRL/G</td>
<td>DCL</td>
</tr>
<tr>
<td>DEBUG</td>
<td>DECSIM.INI</td>
<td>DETECT</td>
<td>DISABLE</td>
</tr>
<tr>
<td>DROP</td>
<td>EDIT</td>
<td>EDITCOMMAND</td>
<td>ENABLE</td>
</tr>
<tr>
<td>ERROR</td>
<td>Escape_Prompt</td>
<td>EVALUATE</td>
<td>EXAMINE</td>
</tr>
<tr>
<td>Expressions</td>
<td>FAULT</td>
<td>FILE_SPEC</td>
<td>EXAMINE</td>
</tr>
<tr>
<td>HALT</td>
<td>HELP</td>
<td>IF</td>
<td>FORMAT</td>
</tr>
<tr>
<td>Label</td>
<td>LEAVE</td>
<td>INDIREC</td>
<td>GRIP</td>
</tr>
<tr>
<td>Machine_desc</td>
<td>MACRO</td>
<td>LOAD</td>
<td>GRIP</td>
</tr>
<tr>
<td>New_release</td>
<td>MOS</td>
<td>LOG</td>
<td>GRIP</td>
</tr>
<tr>
<td>PATTERN</td>
<td>New_user</td>
<td>Networks</td>
<td>KEYpad</td>
</tr>
<tr>
<td>PAUSE</td>
<td>NOTES</td>
<td>New_features</td>
<td>KEYpad</td>
</tr>
<tr>
<td>REPLAY</td>
<td>Prefixes</td>
<td>Numbers</td>
<td>Options</td>
</tr>
<tr>
<td>RESTORE</td>
<td>PRINT</td>
<td>Operators</td>
<td>Options</td>
</tr>
<tr>
<td>SHARE</td>
<td>RESUME</td>
<td>Prompts</td>
<td>OPTIONS</td>
</tr>
<tr>
<td>STATISTICS</td>
<td>SAVE</td>
<td>RADIX</td>
<td>RECALL</td>
</tr>
<tr>
<td>STEP</td>
<td>SPAWN</td>
<td>SCOPE</td>
<td>SELECT</td>
</tr>
<tr>
<td>System_variable</td>
<td>SUBSCRIPTS</td>
<td>STATE</td>
<td>SET</td>
</tr>
<tr>
<td>UNLOAD</td>
<td>TERMINAL</td>
<td>SYMBOLE</td>
<td>SYNONYM</td>
</tr>
<tr>
<td>UNTIL</td>
<td>USE</td>
<td>SYMBOLE</td>
<td>SYNONYM</td>
</tr>
<tr>
<td>WAIT</td>
<td>VAXSTATISTICS</td>
<td>TIME</td>
<td>TRACE</td>
</tr>
<tr>
<td>WATCH</td>
<td>WHILE</td>
<td>TIMING</td>
<td>TRACE</td>
</tr>
<tr>
<td></td>
<td>ZYCAD</td>
<td>TRACED</td>
<td>TYPE</td>
</tr>
</tbody>
</table>
IF

Provides conditional command execution.

FORMAT

[label:] IF test_condition THEN consequence_action
  [ ELSE alternative_action ]

Default Qualifiers
None.

Optional Qualifiers
None.

restrictions

- You may not abbreviate the keywords IF, THEN, and ELSE.

DESCRIPTION

The IF command allows a sequence of commands to be executed if a given test_condition proves true. If that test_condition proves false, DECSIM ignores the sequence of commands. By using the ELSE command, you can direct DECSIM to an alternative sequence of commands when the test_condition proves false.

The test_condition can be any DECSIM command expression. DECSIM evaluates it when the IF command is executed. The result of the evaluation should be a one bit value (1 for True and 0 for False). If the test_condition evaluates to a U or Z value, DECSIM issues an error message and executes the LEAVE ALL command.

The sequence of commands executed as a result of using IF can include any DECSIM commands, including IF.

ELSE clauses are paired with the closest previous IF statement. Because ELSE is optional, you may need to use line continuation.

A BEGIN-END block should surround a sequence of statements if more than one statement is used as the consequence_action or the alternative_action.

For more information see:

FOR       WATCH
UNTIL     WHILE
EXAMPLES

SIM> IF (i EQ j) THEN BEGIN EXAM mem[0]; SIM; END
MORE> IF (a NEQ 0) THEN
MORE> BEGIN
MORE> PRINT 'ERROR in AREG'
MORE> EXAMINE mem
MORE> END - !Dash is required to show continuation
MORE> ELSE SHOW STATE

Control Flow Tracing

The following tracing is shown when the "DEBUG / LOOP" command is in effect.

SIM> IF 1 THEN PRINT '***THEN***' ELSE PRINT '***ELSE***'
[ IF_TEST_COND is TRUE ]
***THEN***
SIM> IF 0 THEN PRINT '***THEN***' ELSE PRINT '***ELSE***'
[ IF_TEST_COND is FALSE ]
***ELSE***
SIM> IF 0 THEN PRINT '***THEN***'
[ IF_TEST_COND is FALSE ]
[ No ELSE alternative specified ]
SIM>
INDIRECT (@)

The INDIRECT (@) command allows you to execute a text file that contains commands. DECSIM reads the file and executes the commands as if they were typed interactively.

The INDIRECT variants prefixed with SET, SHOW, CANCEL, ENABLE, and DISABLE are described as separate commands.

The SET INDIRECT and @ commands (equivalent commands) change the source of DECSIM command input, allowing DECSIM to accept commands from a disk file.

FORMAT

[SET] INDIRECT [/qualifier...] ind_file [/qualifier...] @ [/qualifier...] ind_file [/qualifier ...]

Default Qualifiers
/NOECHO
/REPEAT=1

Optional Qualifiers
/ECHO
/PAUSE
/REPEAT=Integer

restrictions
None.

ARGUMENTS

ind_file
The name of the indirect file. You can specify one file. The default file extension is .IND.

QUALIFIERS

/ECHO
/NOECHO

Display each line of the file on your terminal as it's processed. Must be given at each level of indirection. If /ECHO is given at the top level, then the default is still /NOECHO for files called from within the first file. If the top-level file has /NOECHO, it is still possible for files called by it to have /ECHO.

To record the indirect file in a log file, enable both INDIRECT/ECHO and LOG/ECHO.

If DEBUG/EXPAND is in effect, macro expansions are displayed on the terminal even if the /NOECHO qualifier has been selected.
/PAUSE
Interrupts reading the file before the first command is read or executed, placing command input at the terminal. This has the effect of an automatic DISABLE INDIRECT, and allows you to single-step through the indirect file by using the CONTINUE command. See also the PAUSE and CANCEL commands.

/REPEAT=integer
Executes the command file a total of integer times. /REPEAT cannot be used in conjunction with concatenated files. Absolute times within repeated files can cause problems unless you take the proper precautions. Qualifiers may be placed after the command or after the ind_file.

DESCRIPTION
The SET INDIRECT and at-sign (@) designate an indirect file as the DECSIM command input source. DECSIM treats the commands in an indirect file as if they were typed at the terminal. DECSIM accepts indirect files anywhere it would normally accept a command.

Note: Macro expansions are also forms of indirect input and can be controlled with the CANCEL, SHOW, ENABLE, and DISABLE variants of the INDIRECT command.

If an indirect file contains incomplete syntax, such as an unended macro definition or quoted string without closing quotes, the file reads in, and DECSIM prompts you with the MORE> prompt.

As in interactive mode, DECSIM reports command errors when it executes indirect files—however, it also reports the line containing the syntax error.

Note: CTRL/Z and CTRL/C are not recognized from within indirect files.

For more information see:
CONTINUE(a command macro)
PAUSE (a command macro)

EXAMPLES
SIM> INDIRECT TEST1
SIM> @X
SIM> @/ECHO INIT
SIM> INDIRECT TEST1.IND.2  !To get a particular version of a file
SIM> @ 'TEST1.IND;2'       !Whole name must be quoted because of the ";".

5–113
CANCEL INDIRECT

Terminates one or more levels of indirect command files or command macros and returns to the next level (upward).

**FORMAT**

CANCEL INDIRECT [/qualifier... ]

Default Qualifiers
/LEVELS=1

Optional Qualifiers
/ALL
/LEVELS=integer

**restrictions**

None.

**QUALIFIERS**

/ALL
Terminates all levels of indirect files and returns to terminal input.

/LEVELS=integer
Specifies number of levels to be terminated.

**DESCRIPTION**

CANCEL INDIRECT cancels pending indirect files or command macros, and lists them on your terminal. By default, it cancels one level at a time.

This command can be used in an indirect command file (as an action of an IF statement, for instance), or may be typed from the terminal while a PAUSE is in effect.

An additional command is needed to exit from a control flow block. Because DECSIM reads in all commands within a control flow block at once, using the CANCEL INDIRECT does not interrupt the flow of commands, although it does cancel the indirect file. Thus, you should follow the CANCEL INDIRECT command with the LEAVE ALL command.

For more information see:

LEAVE

**EXAMPLE**

SIM> CANCEL INDIRECT /LEVELS=2

5-114
DISABLE/ENABLE INDIRECT

The DISABLE/ENABLE INDIRECT commands disable or enable command input from a file.

<table>
<thead>
<tr>
<th>FORMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISABLE INDIRECT</td>
</tr>
<tr>
<td>ENABLE INDIRECT [/qualifier]</td>
</tr>
</tbody>
</table>

Default Qualifiers
None.

Optional Qualifiers
/NEXT [=integer]

restrictions
None.

QUALIFIERS
/NEXT [=integer]
ENABLE INDIRECT/NEXT allows you to read and execute an indirect file, command by command, pausing where you specify. /NEXT integer tells DECSIM how many top-level command statements to read and execute before it pauses. /NEXT echoes the command statements. The default for /NEXT is 1.

DECSIM treats BEGIN-END and control blocks as one command.

You can use ENABLE INDIRECT/NEXT (or CONTINUE/NEXT) to single step through an indirect file, opened with the @/PAUSE command.

DESCRIPTION
The DISABLE INDIRECT command (or PAUSE command macro) switches command input from an indirect file or macro to the terminal. The indirect file or macro that was in progress is left suspended until you enable or cancel it. While the file is suspended, you can invoke commands or macros, or read additional indirect files.

DISABLE INDIRECT (or PAUSE) commands contained in control flow blocks are executed after the control loop is complete, because DECSIM parses the entire control flow command before executing any of it.

For more information see:
CONTINUE (a command macro)
SIMULATE/CONTINUE
PAUSE (a command macro)
DISABLE/ENABLE INDIRECT

EXAMPLES

SIM> DISABLE INDIRECT

Entered from an indirect file or a WATCH action macro expansion.

SIM> ENABLE INDIRECT

Entered from the terminal.
SHOW INDIRECT

Prints the names of all indirect command files and macro expansions that are in progress.

**FORMAT**

<table>
<thead>
<tr>
<th>SHOW INDIRECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Qualifiers</td>
</tr>
<tr>
<td>None.</td>
</tr>
<tr>
<td>Optional Qualifiers</td>
</tr>
<tr>
<td>None.</td>
</tr>
</tbody>
</table>

**restrictions**

None.

**DESCRIPTION**

In addition to listing all indirect command files that are in progress, SHOW INDIRECT lists the complete file specifications on your terminal. It also lists the level numbers of nested files.

**EXAMPLE**

SIM> SHOW INDIRECT
INFORM (Command Macro)

INFORM (Command Macro)

Allows you to pass a message from your terminal or from an indirect file to your terminal, and to all open log files except for TRACE log files.

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>INFORM 'anytext'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INFORM text</td>
</tr>
</tbody>
</table>

Default Qualifiers
None.

Optional Qualifiers
None.

restrictions
None.

DESCRIPTION
DECSIM treats all text between INFORM and the carriage return or second quotation mark as a comment. Unlike comments that begin with "!" however, the INFORM comment is printed on your terminal. INFORM is useful in any input file.

For more information see:
COMMENT LOG PRINT

EXAMPLES

SIM> INFORM through with memory loading

INFORM Macro Definition

```
SET MACRO /COMMAND/ABBREV=3 INFORM [$TEXT] =
  PRINT '!' ',', $TEXT'
$ENDMACRO
```
KEY

Associates an equivalence string and a set of attributes with a key on the terminal keyboard. To enable the KEY command, you must use the following VMS command.

SET TERMINAL/APPLICATION

**FORMAT**

[SET] KEY  [/qualifier...] keyname 'equivalence_string'

Default Qualifiers
/ECHO
/LOG
/NOLOCK_STATE
/TERMINATE

Optional Qualifiers
/IF_STATE=(state_name,...)
/LOCK_STATE
/NOECHO
/NOLOG
/NOTERMINATE
/SET_STATE=statename

**restrictions**

- You must execute the VMS command SET TERMINAL/APPLICATION in order to enable the KEY command.

**ARGUMENTS**

Equivalence_string is a text string that is processed when you press the key.

Keyname is one of the following keys, which cannot be abbreviated:

- KP0 – KP9
- PF1 – PF4
- ENTER, PERIOD, COMMA, MINUS
- Left Arrow and Right Arrow
- Find (E1), Insert Here (E2), Remove (E3), Select (E4), Prev Screen (E5), Next Screen (E6)
- Help, Do
- F6 – F20
QUALIFIERS

/ECHO
/NOECHO
Determines whether the equivalence string is displayed on your screen after the key has been pressed. The default is /ECHO. You cannot use /NOECHO with the /NOTERMINATE qualifier.

/IF_STATE=(state_name,...)
Specifies a list of one or more states, one of which must be in effect for the key definition to be in effect. If you omit the /IF_STATE qualifier, the current state is used. The state name is an alphanumeric string. States are established with the /SET_STATE qualifier or the KEY command. If you specify only one state name, you can omit the parentheses. By including several state names, you can define a key to have the same function in all the specified states.

/LOCK_STATE
/NOLOCK_STATE
Specifies that the state set by the /SET_STATE qualifier remain in effect until explicitly changed. If you use the /NOLOCK_STATE qualifier, the state set by /SET_STATE is in effect only for the next defineable key that you press, or for the next read terminating character that you type.

The default is /NOLOCK_STATE. The /LOCK_STATE qualifier can only be specified with the /SET_STATE qualifier.

/LOG
/NLOG
Controls whether the system displays a message indicating that the key definition has been successfully created. The default is /LOG.

/SET_STATE=statename
Causes the specified state name to be set when the key is pressed. The state name can be any alphanumeric string.

If you omit the SET_STATE qualifier or use /NOSET_STATE, the current state that was locked remains in effect. If you have not included this qualifier with a key definition, you can use the SET KEY command to change the current state.

/TERMINATE
/NOTERMINATE
Specifies whether the current equivalence string is to be terminated (that is, processed) when the key is pressed. The default is /NOTERMINATE, which allows you to press other keys before the equivalence string is processed. Pressing RETURN has the same effect as using /TERMINATE.

The /NOTERMINATE qualifier allows you to create key definitions that insert text into command lines, after prompts, or into other text that you are typing.
The default keypad follows:

```
| GOLD | HELP   | SHOW | SET | : default def'n
| RECALL | SYMBOLS* | LABELS* | CANCEL | : GOLD def'n
| USE | LOG | PATTERN | VMS | ---------
| COMPIL | MACRO | LOAD | QUIT* | ---------
| FAULT | CALL | TRACE | SAVE | ---------
| DETECT | ACTIVATE | WATCH | RESTORE | ---------
| EXAMINE | EVALUATE | DEPOSIT | | ---------
| TIME | RADIX | SCOPE | ENTER | ---------
| SIMULATE | RESUME* | | | ---------
| SIMULATE* | STEP | | | ---------
```

* denotes key terminates; other keys expect more input before <CR>

Commands can be constructed; e.g., PF3***P9 = SHOW PATTERN

---

**EXAMPLE**

SIM> SHOW KEY pf4
DEFAULT keypad definitions:
   PF4 = 'SET'
SIM> KEY pf4 'DEPOSIT'
  %I-DEFKEY, DEFAULT key PF4 has been defined
SIM> SHOW KEY pf4
DEFAULT keypad definitions:
   PF4 = 'DEPOSIT'
SIM> KEY/SET_STATE=GOLD/NOTERMINATE/ECHO pf1 'SHOW'
  %I-DEFKEY, DEFAULT key PF1 has been defined
SIM> SHOW KEY PF1
DEFAULT keypad definitions:
   PF1 = 'SHOW'
SIM> KEY/TERMINATE_IF_STATE=GOLD/ECHO pf1 'SET'
  %I-DEFKEY, GOLD key PF1 has been defined

---

5-121
SHOW KEY

SHOW KEY

Shows the current key definitions. To use this command, your terminal must be set up in applications mode. Use the following VMS command:

SET TERMINAL/APPLICATION

FORMAT

SHOW KEY [/qualifier] [keyname]

Default Qualifiers

/BRIEF

Optional Qualifiers

/ALL
/FULL
/PAD
/STATE

restrictions

• You must execute the VMS command SET TERMINAL/APPLICATION in order to enable the KEY command.

ARGUMENTS

keyname

Keyname is one of the following keys, which cannot be abbreviated:

• KP0 – KP9
• PF1 – PF4
• ENTER, PERIOD, COMMA, MINUS
• Left Arrow and Right Arrow
• Find (E1), Insert Here (E2), Remove (E3), Select (E4), Prev Screen (E5), Next Screen (E6)
• Help, Do
• F6 – F20

QUALIFIERS

/ALL
Shows all the defined keys.

/BRIEF
Reports only the definition of the specified key.

/FULL
Reports the definition of the specified key and its attributes (see the KEY command).
SHOW KEY

/PAD
Displays a picture of the current keypad.

/STATE
Reports the key definition with the state (see the KEY command).

EXAMPLE

SIM> SHOW KEY PF4
  DEFAULT keypad definitions:
  PF4 = 'SET'
SIM> SHOW KEY/FULL PF4
  DEFAULT keypad definitions:
  PF4 = 'SET' (echo, noterminate, noerase)
SIM> SHOW KEY/BRIEF PF4
  DEFAULT keypad definitions:
  PF4 = 'SET'
LABELS (Command Macro)

Calls the EXAMINE command to print all the labels at the current hierarchical level, as defined by the SCOPE command.

**FORMAT**

```
LABELS
```

Default Qualifiers

```
None.
```

Optional Qualifiers

```
None.
```

**restrictions**

```
None.
```

**DESCRIPTION**

The LABELS command macro is the equivalent of the command EXAMINE/LABELS *.

For more information see:

```
EXAMINE/LABEL
```

**EXAMPLES**

```
SIM> USE TSADDER
    Reading File: DISK$DECSIM:[FILLMORE]TSADDER.NOB;4
    Time Resolution = 1 NS, default units is nS

SIM> LABEL
    D1: Model = SADDER Revision = 0

SIM> SET SCOPE D1

SIM> LABEL
    C1: Model = SBNA Revision = 0
    C2: Model = SBNA Revision = 0
    C3: Model = NOT Revision = 0
    C4: Model = NOT Revision = 0
    C5: Model = NOT Revision = 0
```

**LABELS Macro Definition**

```
SET MACRO/COMMAND/ABBREV=3 LABELS = EXAMINE/LABELS "*" SENDMACRO
```
LEAVE

Allows you to exit from the middle of a control flow block or BEGIN-END block.

**FORMAT**

LEAVE label  
LEAVE ALL

**Default Qualifiers**

None.

**Optional Qualifiers**

None.

**restrictions**

- You may not abbreviate the keyword LEAVE.

**DESCRIPTION**

The LEAVE label command allows you to exit from a labeled control flow block (IF, FOR, WHILE, UNTIL, SELECT) or BEGIN-END block. Command execution resumes at the next command following the labeled block.

A LEAVE command can be included in any block or iterative command. It may be given from the terminal, an indirect file, or a WATCH command. It allows you to exit from the block containing it.

The LEAVE command must be within the scope of the block or command labelled by the LEAVE label. That is, a LEAVE from one WATCH command cannot reference a label defined in another WATCH command.

Labels are global. Two labels of the same name cannot be used at the same time. However, once a block or statement is exited (or a WATCH canceled), the label name can be reused. The predefined label ALL can be used in the LEAVE command to exit to the top-most level of the current context. The following rules apply:

- If the commands were entered interactively from the terminal, a LEAVE ALL will return you to command mode. A standard prompt will appear as if the command sequence had been completed normally.

- If an indirect file or macro was executing, control flow will return to the next top-level command contained in the file or macro. If that was the end of the file or macro, the prompt will appear.

- If the LEAVE ALL was requested from within a WATCH action command, the control flow resumes at the next "top-level" WATCH action command. If there are no more in that WATCH action, the WATCH action is completed.

- If a LEAVE is given within an iteration command, the iteration command is terminated at the point of the LEAVE.
LEAVE

For more information see:

FOR UNTIL
IF WATCH
SELECT WHILE

EXAMPLE

Control Flow Tracing

The following tracing is shown when the "DEBUG/LOOP" command is in effect.

SIM> L1: IF 1 THEN
MORE> IF 1 THEN LEAVE L1
[ IF_TEST_COND is TRUE ]
[ IF_TEST_COND is TRUE ]
[ LEAVE L1 ]
SIM>
LOAD

Reads data from a file or terminal to store into a command language state, behavior state, structural memory, or ZYCAD memory.

FORMAT

LOAD/FILE=filename [/qualifier... ] state_name [address_range]
LOAD [/qualifier... ] state_name [address_range]

Default Qualifiers
FILE=state_name.MEM
/NOCHECK
/TEXT

File Format Qualifiers
/EXECUTABLE
/MICRO

Optional Qualifiers
/CHECK
/FILE=filename
/INITIAL=integer

restrictions
None.

ARGUMENTS

state_name
Name of data to be loaded (can be a behavior or command state, or structural, ZYCAD, or soft DECSIM memory).

address_range
The address [integer] or range of addresses [i:j] into which to read the data. The loading of data into the array is offset, using the beginning subscript as a base. The loading stops when the ending subscript is reached. In a TEXT file where addresses are also specified in the file, loading is restricted to those addresses that fall within the range specified in the LOAD command.

If no address range is specified, loading begins at the base address.

QUALIFIERS

/CHECK
/NOCHECK
Compares the word subscripts of the specified state array with the word subscripts specified in the LOAD command to determine whether the loaded file will exceed the bounds of the array.

/NOCHECK is the default.
/EXECUTABLE
Places information from a VMS binary-formatted image (.EXE) file, into the specified command language state, behavior state, structural memory, or ZYCAD memory. The source program, from which the image is created, is generally written in the VAX Macro language. In order to prepare a Macro program to be loaded, follow this procedure:

1 Insert the following PSECT statement in your source code:

```
PSECT $CODE, EXE, NOWRT, PAGE
```

This statement indicates that your macro program is code that represents READ/WRITE, page-aligned memory.

2 Assemble the program with the MACRO–32 assembler by using the DCL command MACRO.

**Format**

MACRO filename.MAR [/LIST]

The /LIST qualifier produces a macrocode listing file where the object code and addresses are shown beside the corresponding source code.

3 Link the program with the DCL command LINK/SYSTEM.

**Format**

```
LINK/SYSTEM=base_address filename
```

The /SYSTEM qualifier creates a binary file, which is in a format that DECSIM can read. This file has no formatting blocks or zero suppression, but contains a series of contiguous, sequential longwords.

The base address is usually set to zero (0), which indicates the linker is to create an executable image with no offset. See the VAX/VMS Linker Reference Manual for more information about the LINK/SYSTEM command. Also, see Step 6 for a description of how the base address used to link the program corresponds to the way you enter the LOAD command.

4 From within DECSIM, bring in your network with the USE or COMPILE command.

5 If applicable, use the SCOPE command to go down to the hierarchical level of the routine that contains the array representing memory.

6 Enter the LOAD command with the qualifiers /EXECUTABLE and /FILE, and specify the name of the array that is being loaded.

**Format**

```
LOAD/EXECUTABLE/FILE = filename array_name
```

Here are three cases of how to LINK and LOAD your program into an array.

a. If you want no offset, you link the program with the base address specified as zero (0), and place the code into the array with no subscripts on the array name.

**Example**

```
LOAD/EXECUTABLE/FILE = startup long_array
```
b. If you are offsetting the placement of the binary code into the array, you still link the program with the base address specified to zero (0). However, you indicate the subscripts of the beginning address index and the ending address index (or the largest declared index) in the LOAD command. In the following example, the array is declared to have 32-bit wide longwords, which correspond to the VAX hardware. When you load the array, you may want to load it according to a virtual address space, which is addressed in bytes, not longwords. Therefore, to get the byte address to correspond to the longword address, divide the beginning address index (and the ending address index) by 4.

Format

LOAD/EXE/FILE = file_name array_name[ offset : endword ]

Example

LOAD/EXE/FILE = filename array[ 4000 #16/4 : FFFF #16/4 ]

If you had declared the array to be 8-bits wide, you would not need to perform the division. If you had declared the array to be a quadword, 64-bits wide, you would divide it by 8.

c. If you do not specify the base address to be 0 when you link your program, you must know the physical memory address of the specified base address when you load the code. By default, /SYSTEM sets the virtual address to %x80000000. If this address, for example, corresponds to the physical byte address 200#16, then you use the following LOAD command to place the code into the array starting from the first longword:

Example

LINK/ SYS= %x80000000
LOAD/EXE/FILE = filename array[ 200 #16/4 : FFFF #16/4 ]

7 Apply stimuli and simulate as usual.

Example

This example is a simple macro code program that sets the current address to 200#16, adds 1 to 2, and places the result in R0.

.PSECT $CODE, REL, EXE, RD, PAGE 
  ^=^x200
START: ADDL3 #1, #2, R0
  !Locate code in PSECT called $CODE
  !Start with PC = 200
HALT
  !R0 <-- 1 + 2
.END START

Note: The transfer (start execution) address is not used in LOAD/EXECUTABLE because it is an unformatted file. DECSIM has no explicit conventions around transfer addresses. You must EVALUATE PC = start explicitly.

/FILE = filename

Specifies the file that is to be read in to initialize the memory model. The file name default is (STATE) state-name, and the extension default depends on the format of the file as given by one of the qualifiers in this list. If SYS$INPUT is specified as the file name, then DECSIM prompts with a LOAD> prompt and waits for you to type load data. To end data input from SYS$INPUT, type CTRL/Z.
/INITIAL=integer
Supplies a value to be loaded in specified address locations. If the address range is supplied, only that range is initialized; by default, all declared contents receive that initial value. The initial value is applied after the file is opened and before the file is loaded.
Known bug: This qualifier cannot be used in conjunction with the /MICRO qualifier; the command is ignored.

/MICRO
Specifies that a file generated by MICRO2 is loaded into behavior states. The data in the file is broken into 32-bit chunks. The symbolic information is not loaded. The default extension is .ULT. See the LOAD/MICRO File Format section below for more information.
In ZYCAD mode, this format can be used to load a ZYCAD Functional Memory.
Known bug: This qualifier cannot be used in conjunction with the /INITIAL qualifier; the command is ignored.

/TEXT
Specifies that the file is an ASCII file. The default extension is .MEM. See specific text file formats in the next section. The radix mark (#) is valid on any number in a text file.

DESCRIPTION
The LOAD command can load data into command states defined at runtime, as well as behavior states in the model definition file that were defined at compile time. If a command state and a behavior state have the same name, the command state is loaded.

The LOAD command reads a variety of disk file formats (both binary and ASCII) and initializes the contents of part or all of a storage element. This will overwrite the current contents of the state storage or memory model. The name of every state loaded is printed on the terminal.

For more information see:
ALLOCATE UNLOAD
DEPOSIT EVALUATE

LOAD/MICRO File Format
The /MICRO qualifier specifies that MICRO2 generated the file, which is called a ULD file. The default file extension is "ULT". See the MICRO2 USER'S GUIDE (AA-H531A-TE, June '79) for a complete description of the format. Briefly, the subset recognized here consists of the code and header statements.

Header Statements in a .ULT File
The header statements are of the form:

;RTOL
or ;RADIX radix-in-decimal
Any other statement beginning with a semicolon (;) is treated as a comment and is ignored without warning. Spaces, tabs, and form feeds are ignored.

All addresses and data are interpreted in the current radix as set by the ;RADIX statement. The default radix is base 8.

Note: Symbol definitions in the .ULD file will be ignored without warning by the LOAD command.

Code Statements in a .ULD File

The code statements are in the form:

[address] single-letter-mem-name = data

The data can be up to 128 bits wide and is padded with zeros, if necessary, to maintain the correct number of digits in the current radix. If the memory name references a nonexistent memory, the loading of the ULD file terminates with an error.

An Example of Loading .ULD Files

DECSIM loads the data directly into the memory named xCODE, where x is the single letter memory name in the .ULD file. The following ULD file is loaded:

;RADIX 16
[0B]U = 41A00A500D
[09]U = 4920090000
[0A]U = 40A00B0000
[4]A = 3535
[0]M = 11112222333344445555666677778888
[1]M = 99999999BBBCCCCDDDDDEEEEFFF0000

The following states are where the file is loaded:
STATE UCODE[0:10]<39:0>;
STATE ACODE[0:6]<15:0>;
STATE MCODE[0:20]<127:0>;

The following memory locations are therefore loaded (all numbers are in hexadecimal):
UCODE[9] = 4920090000
UCODE[A] = 40A00B0000
UCODE[B] = 41A00A500D
ACODE[6] = 7575
ACODE[4] = 3535
MCODE[0] = 11112222333344445555666677778888
MCODE[1] = 99999999BBBCCCCDDDDDEEEEFFF0000
LOAD

LOAD/TEXT File Format

The /TEXT qualifier specifies that the file to be loaded is an ASCII file containing two types of text file statements: switches and address-data. These, along with syntax conventions, are discussed in the following sections.

Text File Commands

These commands are included within the text file and are not part of the LOAD command line. There are two types of qualifiers available: /NAME and /RADIX. These are described in detail below.

Note: The colon (:) here is required as shown and may not be replaced with an equal sign (=).

Qualifiers

/NAME:state_name
determines which particular state is to be loaded in subsequent statements. Like /RADIX, a particular /NAME qualifier is in effect for every statement following the qualifier and before the occurrence of another /NAME qualifier. If an invalid name is given in a /NAME command, a warning is given. The loading stops only if you try loading data when you do not have a valid state to load into.

State_name is the state name you want. You may enclose the state name in single quotation marks. You must enclose the state name in quotes if there are any embedded blanks, that is, spaces or tabs, in the name.

/RADIX:integer
Integer represents a decimal integer in the range from 2 to 16. Specifies the radix to be used in interpreting the numeric values in load statements following the qualifiers and before the occurrence of another RADIUS qualifier.

EXAMPLES

!File name: TEXT.MEM
/RADIX:10
[1]:11
[2]:22
[3]:33
[4]:57,66 !Note that [5] = 66
[6]:77

Text File General Syntax and Conventions

Comments, as well as blank lines, may be placed anywhere in the file. The format for a comment is:

!comment text

Whenever an exclamation point (!) is seen in the file, the rest of the line is considered to be a comment and is ignored.
Spaces and tabs can be inserted to increase the readability of the /TEXT file. In the /NAME qualifier, all spaces, tabs, and blank lines are ignored except between single quotation marks and within a numeric or keyword lexeme. Separate address–data statements are separated by a comma (,) or semicolon (;). Comments always extend to the end of the line, not to the next comma or semicolon.

Statements in the Text File: Address–Data Specification

There are several different types of the load statement: implicit, simple, range, wildcard, and composite.

The Implicit Address Statement

The implicit load statement consists of a single data integer (in the current radix) and specifies that the integer value is to be loaded at the "current" address, after which the current address is to be incremented by one. The current address is initially set at the lowest address of the state being loaded, or it is (re)set to the low address in the address range if one is given in a /NAME qualifier or the command line.

The Simple Address–Data Load Statement

The simple load statement has the form:

```
[address]:data
```

where address is to be loaded with data. After this statement is executed, the current address is incremented. Contiguous blocks of memory can be loaded with sequences in the form:

```
[start–address] : first–data
second–data
...
last–data
```

Using [start–address]: is equivalent to /NAME:[start–address] for the purpose of setting the "current" address.

Unreferenced and Multireferenced Addresses

If an address location is specified more than once, the last value will be kept. If an address location is never referenced, its value remains untouched unless the /INITIAL qualifier is specified in the command line. In this case, all the locations in the specified address range are initialized to the given value.
LOG

Controls opening and closing of files (and other terminals) for outputting both command information and signal activity.

The LOG variants prefixed with SHOW, CANCEL, ENABLE, and DISABLE are described as separate commands.

SET LOG

The SET LOG command opens a file for output. Use the log file facility to:

- To capture a transcript of the terminal input and output
- To capture trace or hazard information in a file
- To duplicate the screen of one terminal session on another screen
- To create an indirect file of commands for later replay or as a journal file

FORMAT

[SET] LOG  [/qualifier...] file [/qualifier... ]

Default Qualifiers

/HEADER
/LOGNAME=filename
/NOCOMMANDS
/NODETECT
/NOECHO
/NOHAZARD
/NOVT
/NOSHARE
/NOTRACE
/RESPONSES

Optional Qualifiers

/APPEND
/CHECKPOINT
/COMMANDS
/DETECT
/ECHO
/LOGNAME=logname
/HAZARD
/NOHEADER
/NORESPONSES
/SHARE
/TITLE='title_string'
/TRACE
/BATCH
/LA36
/LA120
/VT05
/VT50
/VT52
/VT100
Log files never capture system messages printed on the terminal during a network compilation. In other words, errors from NETPRO and SX compilations and system BROADCAST messages are not written to a LOG file.

- The maximum number of log files that can be opened in a DECSIM session is 29.

**ARGUMENTS**

**file**

File is only a single file spec and may not contain concatenations and wildcards.

**Note:** Choose file names with care. DECSIM file names may be confused with VAX/VMS logical devices. For example, if you give the DECSIM command "LOG TT" you will not get a file named TT.LOG, but rather one named TTn: (the user's terminal).

**QUALIFIERS**

/APPEND

Appends to an existing file. If the file does not exist, then a new file with that name is opened. Unless otherwise specified, the default qualifiers are used, regardless of what qualifiers you set in the original LOG command. Specify the file name, not the log name, on the command line.

**Note:** The RESTORE command closes all LOG files, then reopens them in the APPEND mode.

/CHECKPOINT

Writes file to disk, but leaves file open. /CHECKPOINT is useful for protecting output in long simulations against system crashes. /CHECKPOINT is equivalent to the sequence:

CANCEL LOG/CLOSE
LOG/APPEND [with the same qualifiers as listed in the LOG command]

If you want a checkpoint to occur at regular intervals during simulation (say, every 10000 ns), the WATCH /EVERY=time command will evoke the LOG/CHECKPOINT feature. In this example, time would equal 10000. This feature should not be used in conjunction with /TTY.

**Note:** On the VAX, you cannot read a file that is open for writing. Thus you cannot checkpoint a log file in order to type it within DECSIM.

/COMMANDS

/NOCOMMANDS

Copies text of all DECSIM terminal stream input into file. You can use this qualifier to record the commands you enter for later playback as an indirect command file. /NOCOMMANDS is the default.

/DETECT

/NODETECT(D)

Determines whether the log file is a DETECT output file. When /DETECT is entered, the /NOCOMMANDS, /NORESPONSES, and /NOECHO qualifiers are used to open a file with the .DET extension. To cancel
this log you must first enter a CANCEL DETECT command. When
/NOEVENT is entered the default file extension is .LOG.

/ECHO
/NOECHO
Controls the recording of indirect file use. If both INDIRECT/ECHO and
LOG/ECHO are enabled, an exact copy of the session is recorded.
LOG/NOECHO disables the recording of indirect file use, but allows
terminal input to be captured in the LOG file.

/HAZARD
/NOHAZARD
Defines a logname and opens a log file for hazard information that is
routed by default only to the terminal screen. The log file is opened with
the /NOCOMMANDS, /NORESPONSES, and /NOECHO qualifiers and has
a default extension of .HAZ. /NOHAZARD is the default.

Note: You must execute the [SET] LOG command before you compile
or load a model that contains hazard checkers into DECSIM. Use
/RESPONSES on the [SET] LOG command; otherwise, the hazard
information defaults to the screen only.

/HEADER
/NOHEADER
Prints the following header at the beginning of the file:

!Network=NETWORK Revision 0 Source file=filename Simulation Clock= |20 nS

The version number on the VAX is in the format version.edit. If no
network is loaded, the second line is not printed. /HEADER is the default.
/NOHEADER is useful for creating files that are to be used for DIFF
comparisons between versions and for creating files that are to be used as
machine-readable input files for follow-on processes.

/LOGNAME=logname
This qualifier creates a label for the log file. You can use this label to
refer to the log file in other SET, CANCEL, ENABLE, DISABLELOG
commands, in TRACE/LOGNAME or PRINT/LOGNAME commands, or in
%PRINT(LOGNAME) behavioral model statements. The default is the file
name from the file specification. A log name has a limit of 25 characters
and must be alphanumerical.

If no file name was given and no /LOGNAME= was supplied, then the
device name is used. This commonly happens with TT: or LP:. Note that
if a second LOG command with a default of this nature is requested, an
error message states that the log name is already in use. In this case,
simply supply a unique log name with this /LOGNAME qualifier.

/RESPONSES
/NORESPONSES
Copies all terminal output to file. A combination of /COMMANDS and
/RESPONSES makes a complete copy of the terminal transactions.
SHARE
NOSHARE (D)
Allows the log file to be read by another process. When log files are read by other processes there is some loss of performance.

TITLE = 'title_string'
Specifies a title to print at the top of every page. A carriage return is always supplied at the end of the string.

TRACE
NOTRACE
Indicates this is (not) a trace LOG file, as opposed to a terminal log file. When /TRACE is selected, a LOG file with /NOCOMMANDS, /NORESPONSES, /NOECHO /HEADER is opened. You can then direct the output of a trace to this LOG file with the TRACE/LOGNAME command. The default extension for the /TRACE qualifier is .TRA.

To open a binary format trace file with the SET LOG command, use the /NOHEADER and /TRACE qualifiers together. Or, you can use the /LOGNAME qualifier on the TRACE command, which opens a log file if none with the specified name is open.

To append information from a trace to an existing binary trace file, execute a SET LOG/TRACE/NOHEADER/APPEND command first, then reference the log file with the /LOGNAME qualifier on the TRACE command.

LA36
LA120
BATCH
VT05
VT50
VT52
VT100
NONVT
Passes video terminal formatting escape sequences to file. NONVT (no escape sequences) is the default.

DESCRIPTION
The SET LOG command opens a file for output. It may be used to record commands entered from the terminal for later playback as an INDIRECT command file, to capture a copy of the terminal interactions, or to record signal traces. The file name must be given, even if it is TT.

The LOG command associates a logical name with a log file. That logical name is the argument to the /LOGNAME qualifier. If you omit the /LOGNAME qualifier, the file_name is used as the logical name. The logical names for log files must be unique.

The other DECSIM commands use this logical name as specified in their /LOGNAME qualifiers to determine the log file to which their information is written.
If two files have different logical names, but the same file specifications, the operating system will choose to keep only the one that is closed (CANCEL LOG/CLOSE) last.

Normally, SET LOG is used to open a file for writing. However, should changes need to be made in the qualifiers after the initial LOG command has been given, a special modify SET LOG command can be used. There is no difference between this and the normal SET LOG commands in syntax. However, in the modify command, the log name must already exist (via a previous nonmodify SET LOG command). The modify mode is used for /CHECKPOINT and also for adjusting /ECHO and /RESPONSES settings during writing of a file.

A LOG /CHECKPOINT /LOGNAME:EXAMP command closes the file and reopens it in append mode. The file still cannot be read concurrently until it is closed with a CANCEL LOG, EXIT, etc. The directory report is shown below. The file revision number "1" is not changed, but the "Revised" date and number (2) are changed at the time of the /CHECKPOINT.

$ DIR
EXAMP.LOG
Directory _DBB2:[SHERWOOD.CODE]
EXAMP.LOG;1  Size: 11/12  Created: 11-MAR-1982 10:12
Owner: [150,064]  Revised: 11-MAR-1982 10:20 (2)
File ID: (5521,8,1)  Expires: <None specified>
File organization:  Sequential
File attributes:  Allocation=12, Extend=0
Record format:  Variable length
Record attributes:  Carriage return
Total of 1 file, 11/12 blocks.

For more information see:
COMMENT PRINT/LOGNAME DETECT
INDIRECT TRACE/LOGNAME PATTERN/LOGNAME

%PRINT [LOGNAME]

EXAMPLES

SIM> LOG TRACE.LST

SIM> LOG/APPEND/COMMANDS/NORESPONSES TEST

The following lines can be included in your DECSIM.INI file to create and purge log files.

SIM> LOG/COMMANDS/RESPONSES/ECHO INPUT OUTPUT.LOG
SIM> LOG/COMMANDS/NORESPONSES/ECHO INPUT.LOG
SIM> VMS 'IF FS$SEARCH ("INPUT*.LOG,-5") .NES. THEN PURGE/KEEP=5 INPUT*.LOG'

5-138
CANCEL LOG

Closes (saves) an output file on disk, or deletes a file currently open, depending on the qualifier.

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>CANCEL LOG</th>
<th>[ /qualifier... ] logname</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Default Qualifiers</td>
<td>/ALL</td>
</tr>
<tr>
<td></td>
<td>Optional Qualifiers</td>
<td>/DELETE</td>
</tr>
</tbody>
</table>

**restrictions**

- You must use the logical name to specify the log name.
- If a log file was opened by a DETECT/LOGNAME or a TRACE/LOGNAME command, you must enter a CANCEL DETECT or CANCEL TRACE command before you can close the log file with a CANCEL LOG command.

**QUALIFIERS**

/ALL
Affects all log names except implicit /TTY= terminal.

/CLOSE
Terminates writing to the file and saves it on disk. If a file with the same name exists, /CLOSE creates a new version of the file. /CLOSE is the default.

/DELETE
Terminates writing to the file and does not store it on disk. This qualifier simply deletes the file.

**DESCRIPTION**

The CANCEL LOG command terminates DECSIM output to a file.

Some DECSIM commands have /LOGNAME qualifiers such as DETECT, PATTERN, PRINT, and TRACE. When these commands and this qualifier are entered a log file is opened. To close these log files you must enter a CANCEL LOG command. The commands DETECT/LOGNAME and TRACE/LOGNAME open log files with the .DET or .TRA extensions. These .DET and .TRA files cannot be closed with a CANCEL LOG command unless you first enter a CANCEL DETECT or CANCEL TRACE command.

Unless the file has been disabled (with DISABLE LOG), an information message is entered as the last line of the file.
CANCEL LOG

For more information see:
CTRL/C QUIT
EXIT SET LOG

EXAMPLES

SIM> CANCEL LOG test
SIM> CANCEL LOG/ALL
SHOW LOG

Prints all currently active LOG file names and their qualifiers on the terminal.

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>SHOW LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Qualifiers</td>
<td>None.</td>
</tr>
<tr>
<td>Optional Qualifiers</td>
<td>None.</td>
</tr>
</tbody>
</table>

| restrictions | None. |

| DESCRIPTION | SHOW LOG prints the log name and file specification of each output file, with the qualifiers. |

<table>
<thead>
<tr>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIM&gt; SHOW LOG</td>
</tr>
</tbody>
</table>
DISABLE/ENABLE LOG

Control writing to the output file.

FORMAT

DISABLE LOG logname,...
DISABLE LOG /ALL
ENABLE LOG logname,...
ENABLE LOG /ALL

Default Qualifiers
None.

Optional Qualifiers
/ALL

restrictions
None.

QUALIFIERS

/ALL
Chooses all LOG files currently open.

DESCRIPTION

DISABLE LOG suspends writing to an output file, even though PRINT commands specify the log name. ENABLE LOG reenables writing to the file.

EXAMPLES

SIM> DISABLE LOG trace1
SIM> ENABLE LOG/ALL
SET MACRO

Allows you to tailor commands to your needs by defining a macro.
The MACRO variants prefixed with SET, SHOW, and CANCEL are described
in the following sections.

FORMAT  

[SET] MACRO  
[/context] [/qualifier] macro_name
[template] = definition $ENDMACRO

Default Context
/TEXT

Optional Context
/COMMAND

Optional Qualifiers
/ABBREVIATION=integer

restrictions
None.

ARGUMENTS

context
The context of a macro determines how and where the macro is used. The
context is selected from one of the following:

/COMMAND
DECSIM recognizes command macros only when they are used in
situations where DECSIM expects a command. They generally contain
blocks of commands.

/TEXT
DECSIM recognizes text macros anywhere in a command line. They
contain text (such as signal names or states) and commands, but not
templates or formals.

macro_name
The macro_name functions as a command that invokes macro expansion.
The macro_name must begin with a letter and must contain only letters
and digits. Up to 25 character names are allowed. Special characters are
not allowed. The macro_name must not be “MACRO”.

Example:
MACRO SUPERDUPERTRACER [...] = ... $ENDMACRO
**template**

The template of a /COMMAND macro is enclosed in square brackets, following the macro_name in the definition.

**Format**

```
[ $parameter [ , ... ] ]
```

The template is a placeholder (formal argument) for which you can substitute other text when you call the macro. The different types of templates are called parameters. They are only valid for command macros (of context /COMMAND).

The template defines the type of parameter and its defaults, and is referenced by DECSIM in providing escape prompting. DECSIM checks the template syntax and does not expand the macro if the template does not match. This eliminates most error checking needed inside a macro definition.

Parameters begin with a dollar sign ($). When defining a macro, separate multiple parameters with a comma.

In calling a macro, you replace the parameters with other text. The text for one parameter is separated from that of another parameter with carriage return (CR) or semicolon (;).

**Note:** When calling a macro, it is recommended that you use the semicolon (;) to separate the arguments. Using a space or CR does not always work.

The maximum number of actual parameters is 50.

Parameters can be any or all of the following:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Replacement at invocation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ACCESSLIST</td>
<td>An access list. Valid characters are A–Z, 0–9, &lt;&gt;, ., [ ], %, , and any character enclosed in single quotation marks. Characters are converted to all capitals; the name is expanded only if the name is valid. Wildcard characters do not work here.</td>
</tr>
<tr>
<td>$EXPRESSION</td>
<td>Any DECSIM command language expression. The expression is evaluated once before macro expansion, which does not occur if the expression is not valid. Evaluation of the expression occurs at the actual parameter site, and then the result is passed.</td>
</tr>
<tr>
<td>$FILE</td>
<td>A file name or list of file names separated by commas. Valid characters are A–Z, 0–9, ., [ ], and any character enclosed in single quotation marks.</td>
</tr>
<tr>
<td>$NAME</td>
<td>A single identifier.</td>
</tr>
<tr>
<td>$NUMBER</td>
<td>Any integer. Valid characters are A–F, 0–9, and radix.</td>
</tr>
<tr>
<td>$QUOTEDTEXT</td>
<td>Any text enclosed in single quotation marks ('). Valid characters are any characters enclosed in single quotation marks.</td>
</tr>
</tbody>
</table>
MACRO

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Replacement at invocation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$TEXT</td>
<td>Any text. Multiline text can be substituted for the $TEXT parameter by enclosing it in single quotation marks, or by using the line continuation character, the dash (-).</td>
</tr>
<tr>
<td>$TIME</td>
<td>Any absolute or relative time expression with optional units. Legal characters are 0–9 and units.</td>
</tr>
</tbody>
</table>

You can use a parameter several times in the template. The matching parameter in the definition of the macro must be followed by "number". So, if $TEXT is used twice in a single template description, the first would be referred to in the macro definition as $TEXT.1 and the second as $TEXT.2. For example,

```
SIM> SET MACRO/COMMAND x [[$TEXT, $TEXT ]] =
MACRO> INFORM $TEXT.1
MACRO> INFORM $TEXT.2
MACRO> $ENDMACRO
```

Note: $NUMBER and $EXPRESSION will not retain the width information of some constants of the form 0#16. The width passed to the macro expansion will be the same as if the value had been expressed in base 10.

**Definition**

The macro definition is the body of the macro, and includes everything between the equal sign and $ENDMACRO keyword. The macro definition is substituted (expanded) when you call the macro.

Note: $ENDMACRO is recognized even if it is contained within quotation marks or comments.

**QUALIFIERS**

`/ABBREVIATION= integer`

Specifies the minimum number of letters of the macro name you must type when you call the macro. Allows you to type an abbreviated version of the macro name. The macro abbreviation overrides all other DECSIM abbreviations.

**DESCRIPTION**

The MACRO facility provides a text substitution mechanism and a means for defining new commands, enabling you to develop a customized command language.

When a macro is being defined, DECSIM stores the text and does not provide escape prompting. Macros are not executed, expanded, or checked for syntax until called. To debug your macro, execute a DEBUG/EXPAND command before calling the macro. See the DEBUG command description in Chapter 5 for more information.

Macro definitions are stacked; thus, each macro name can have several simultaneous definitions. Invoking the macro expands the most recent definition. Canceling the macro deletes the most recent definition.
In MACRO arguments using $NUMBER or $EXPRESSION, a constant is evaluated literally, and then the result is reintroduced into the macro definition. This process loses track of the original width, but does not affect the value of the constant.

Conventions for macro expansion include:

- Formal $Parameters within the definition are enclosed in single quotation marks when expanded.
- Blank lines within a macro definition are included in the expansion.
- To allow correct execution when a macro is expanded within a control statement, a macro which contains a sequence of several commands should have them combined into a single compound command enclosed in a BEGIN–END block.

Note: Macro definition/expansion lines can only be 150 characters long. Exceeding the limit is not flagged as an error, but causes erroneous macro expansion. The 150 character limit includes the total “virtual” line length counting continuations, but not BEGIN–END block compound statements.

To concatenate different text sources in a macro definition, use the two dollar sign ($$) lexical operator. Once the splicing is performed, the dollar sign is not entered into the macro expansion.

You can call a macro from within a macro if the called macro is defined before the first macro is expanded.

**EXAMPLES**

1

SIM> MACRO/COMMAND/ABBREVIATION=3 makelogs [FILE, FILE] =
MACRO> BEGIN
MACRO> LOG/COMMANDS/RESPONSES/ECHO FILE.1
MACRO> LOG/COMMANDS/NORESPONSES/ECHO FILE.2
MACRO> END
MACRO> $ENDMACRO
SIM>
SIM> mak journal;testfile

This example creates a command macro named "makelogs," which opens two log files. Notice that a semi-colon (;) is used to separate the arguments in the call.

2

SIM> MACRO/COMMAND ein [FILE] =
MACRO> edit FILE$$ .ind
MACRO> $ENDMACRO
SIM>
SIM> ein table

This example defines a command macro named "ein". This macro takes a file name parameter, concatenates that file name with the "ind" extension to make a complete file specification, and opens the default editor to edit that file.
These two macros show the difference between the $TEXT and $EXPRESSION parameters. While the "mac1" macro has the behavior we might expect, "mac2" does not. The reason is that $EXPRESSION, in this case I, is evaluated before macro expansion. "I" has no value until the macro is expanded and the FOR loop begins, so a value of 0 is assumed.

For more information see:

CANCEL MACRO    SHOW MACRO
SHOW MACRO

The SHOW MACRO command displays macro definitions.

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>SHOW MACRO  [/context] ['macro_name']</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SHOW MACRO /ALL</td>
</tr>
<tr>
<td>Default Context</td>
<td>/TEXT</td>
</tr>
<tr>
<td>Optional Context</td>
<td>/COMMAND</td>
</tr>
<tr>
<td>Optional Qualifiers</td>
<td>/ALL</td>
</tr>
</tbody>
</table>

restrictions

None.

QUALIFIERS

/ALL

Specifies all defined macros.

DESCRIPTION

The macro name in the SHOW and CANCEL MACRO commands must be enclosed in quotation marks. SHOW MACRO shows all the macro names currently defined. SHOW MACRO 'macro_name' displays the macro definition for all contexts of that macro name. SHOW MACRO/ALL prints the headers and contents of all defined macros.

EXAMPLE

The following example shows the DECSIM predefined macros.

SIM> show macro
    /COMMAND COMB*EH
    /COMMAND CONT*INUE
    /COMMAND DBG
    /COMMAND EDITC*COMMANDMACRO
    /COMMAND EDIT*EXTMACRO

These macros can be abbreviated to the string to the left of the asterisk (*).
... SHOW MACRO

SIM> show macro/command 'symbols'
SET MACRO /COMMAND /ABBREV=3 SYMBOLS =
examine "*" ENDMACRO;

This macro can be abbreviated to 'sym'.

SIM> show macro/all
SET MACRO /COMMAND /ABBREV=4 CONTINUE =
enable indirect $ENDMACRO;
SET MACRO /COMMAND /ABBREV=2 TYPE [ '$FILE' ] =
print %infile($FILE) $ENDMACRO;
SET MACRO /COMMAND /ABBREV=3 INFORM [ '$TEXT' ] =
print '!', '$TEXT' $ENDMACRO;
SET MACRO /COMMAND /ABBREV=3 PAUSE [ '$TEXT' ] =
print '!', '$TEXT' ; disable indirect $ENDMACRO;
SET MACRO /COMMAND /ABBREV=3 LABELS =
examine/labels "=" $ENDMACRO;
SET MACRO /COMMAND /ABBREV=3 SYMBOLS =
examine "*" $ENDMACRO;
CANCEL MACRO

Deletes a macro definition.

FORMAT

CANCEL MACRO [/context] 'macro_name'
CANCEL MACRO [/context] /ALL

Default Context
/TEXT (when used with 'macro_name')

Optional Context
/COMMAND

Optional Qualifiers
/ALL

restrictions
None.

QUALIFIERS

/ALL
 Specifies all defined macros for the given context. (Default is /TEXT.)

EXAMPLES

SIM> cancel macro/command 'symbols'
SIM> show macro/command 'symbols'
%E, [SIM#NF] MACRO name not found for given context for this context
For MOS models, specifies a delay time for the switching of all nodes. MOS can be used at any time during simulation.

**FORMAT**

[SET] MOS [/qualified]
SHOW MOS

Default Qualifiers
/NOUNIT_DELAY

Optional Qualifiers
/UNIT_DELAY=[integer][time_unit]

**restrictions**

- Loading a network can change any delay set with the MOS command.

**QUALIFIERS**

/NOUNIT_DELAY
Computes the delay for each node using the MOS algorithm. This is the default.

/UNIT_DELAY [ = integer ] [ time_unit ]
Specifies a delay time for the switching of all nodes. Time units can be FS, PS, NS, US, MS, or S. If neither integer nor time_unit is specified, the delay defaults to the network resolution. If just the integer is specified, the time_unit defaults to that of the network resolution. Delays cannot be smaller than the network resolution.

**DESCRIPTION**
The SET MOS command allows you to specify the time and the units that DECSIM uses as the delay with which all nodes in the circuit switch. You can change the time unit at any time during your DECSIM session by entering another SET MOS command.

If you do not enter a SET MOS command, the default time unit is the resolution of the network.
EXAMPLE

SIM> SET MOS/NOUNIT_DELAY
SIM> SHOW MOS
FULL DECISION MOS SIMULATION.
SIM> SET MOS/UNIT_DELAY
DEFAULT: Time step has been set equal to the resolution unit.
TIME STEP = 1 S
SIM> SET MOS/UNIT_DELAY = 2
TIME STEP = 2 S
SIM> SET MOS/UNIT_DELAY = 3 S
TIME STEP = 3 S
SIM> SHOW MOS
UNIT-DELAY MOS SIMULATION.
TIME STEP = 3 S
SIM> EXIT
PATTERN

Applies pattern vectors stored in a text file to signal inputs of a simulated network.

With the /COMPARE qualifier, PATTERN can compare the actual simulation values of specified signals with the data stored in a pattern (.PAT) file. See the /COMPARE qualifier description for more details.

**FORMAT**

```
[SET] PATTERN /FILE=file_spec [/global_qualifiers...]
             access_list [/local_qualifier]
SHOW PATTERN pattern_name
CANCEL PATTERN pattern_name
DISABLE PATTERN pattern_name
ENABLE PATTERN pattern_name
```

Required Qualifiers

/FILE=file_name

Default Qualifiers

/DELIMITER=SPACE, TAB, FF, and CRLF
/FREE_STATE=?#2
/RADIX=2
/RELATIVE
/NOFINAL_RELEASE

Optional Global Qualifiers

/ABSOLUTE
/BINARY
/COMPARE
/FINAL_RELEASE=argument
/FREE_STATE=logic_state
/LOGNAME=logname
/LOGNAME=(logname,...)
/MAP=(original_name=resulting_name)
/PATTERNNAME=pattern_name
/TIMEUNIT=units
/TOKEN='single_char'
/WIDTH=integer

Optional Local Qualifiers

/DELIMITER='char'
/RADIX=integer
/WIDTH=integer

**restrictions**

- Some restrictions apply in ZYCAD mode. See DESCRIPTION below.
ARGUMENTS

`access_list[/local_qualifier] , access_list[/local_qualifier] ... ]`

The access list must appear on the command line. The access list can contain only signal names, %TIME, and %NULL.

`signal_name`

A list of one or more signals, not used in any other pattern. The PATTERN command can drive network primary inputs (both scalar and vector), equation outputs, and MOS nodes. The pattern command implicitly forces both equation outputs and MOS nodes.

Multibit signals are listed in the access list differently according to the type of model that is instantiated in the network. A multibit signal in a structural model must have an entry in the access list for each bit; the signal name and a 1-bit subscript is entered for each bit. A multibit signal in a behavior model can be listed no more than once in the access list.

`%TIME`

A placeholder, which can appear only once in the access list. It shows the field in the file that contains the time. If no %TIME is given, DECSIM assumes that the first item is the time and issues an informational message. Units of time may not be specified in the pattern file; to specify them enter the /TIMEUNIT qualifier.

`%NULL`

Skips fields in the input file. The fields must still contain valid states (0, 1, U, Z).

GLOBAL QUALIFIERS

Global qualifiers apply to every item in the access list while local qualifiers only affect the item that they follow. Global qualifiers are given before the access list, while local qualifiers are given within the access list.

`/ABSOLUTE`

See the description for /RELATIVE.

`/BINARY`

Specifies that the file specified in your /FILE= file_spec qualifier is in binary format and was created by a TRACE/BINARY command and qualifier. Binary format files contain records of both signal values and signal names.

If your network does not contain the same signal names as the network that was loaded when the TRACE/BINARY command was entered to create the binary format file, you can still use the binary format file as a pattern file. When you do you can handle the signal name difference in one of the following ways:

- Enter an access_list. The order of the items in the access_list is important. DECSIM compares the order of the items in the access_list in the TRACE/BINARY command that created the binary format file to the order of the items in the access_list in the PATTERN/BINARY command. It uses this order to schedule the input of values of signal names in the TRACE/BINARY access_list to signal names in the PATTERN/BINARY access_list.
If the access_list in the TRACE/BINARY command contained a signal whose values you do not want scheduled to be input to any signal in your current network you should enter the %NULL placeholder in that signal's corresponding place in the access_list in the PATTERN/BINARY command.

The PATTERN/BINARY access_list contains no %TIME placeholder or /WIDTH or /DELIMITER qualifier.

- Enter the /MAP qualifier to specify how the values of the signals in the TRACE/BINARY command's access_list are applied to the signals in your current network.

/COMPARE
Compares the actual simulation values of the specified signals with the pattern file data at the times specified in the pattern file. Whenever there is a discrepancy between the pattern file data and the simulation values, DECSIM issues a message that indicates the name of the signal, the time of the discrepancy, the line of the pattern file where the expected value is, the name of the pattern file, and the actual and expected values of the signal. For example:

Signal: SUM1 at Time: 4800 on Line: 6 in Pattern: PATTERN2 Value: 0 Expected: 1

Thus, PATTERN/COMPARE, unlike the PATTERN command without the /COMPARE qualifier, does not drive the pattern file values onto the network signals specified in the access list.

All global and local qualifiers of PATTERN apply to PATTERN/COMPARE. %NULL in the command line is used to skip fields in the pattern file. A question mark (?) in a pattern file vector means "do not compare this value at this time."

/DELIMITER= 'char'
Specifies an additional delimiter character that may appear as the delimiter between signal state values in the vector data. If you specify more than one additional character, only the first will be used. The characters space, tab, and form feed are always delimiter characters. Multiples of the delimiter characters in the file are treated as a single character.

/FILE= file_spec
Specifies the name of a file containing pattern data. The list of signals (access list) to which the state vectors are to be applied must be specified in PATTERN command itself, and may not be contained in the file. The default file extension is '.PAT'. If SYS$INPUT is used as the file name, DECSIM prompts you with the pattern name whenever input would be taken from the pattern vector file. To end pattern input from SYS$INPUT, type CTRL/Z.

Pattern File Format
The data in the pattern file consists of vector_data statements, having the following form:

[ state_list ] time_value [ state_list ]

- state_list – A list of 0 or more signal states, or the question mark (?) signifying "free", possibly separated by delimiters.
* time_value – Either a relative or absolute time value.

In a pattern file, blank lines are ignored. Line continuation is not allowed. You can add comments by preceding them with an exclamation point (!). If you want the comments in a pattern file to be printed to a log file, precede the comment with @. (Example: @ This is a comment.)

A time value must be given on each vector_data statement appearing in a file. It may be specified as relative time with respect to the previous vector_data item, or as absolute time with respect to the simulator clock using the prefix "!".

Signal state values are specified in free format separated by delimiter characters. The default delimiter characters are blanks, tabs, and form feeds. Additional characters can be specified using the /DELLIMITER switch. The state values must be in the radix specified on the command line. If the radix is 2, 4, 8, or 16, the number can contain U's and Z's.

Note: If you do not separate the signal state values in a vector_data statement with delimiter characters, you must enter the /WIDTH qualifier on the command line.

The radix character '#' is not allowed in pattern data.

/FINAL_RELEASE=argument
Specifies what happens to a signal at the end of a pattern file. The argument can be one of the following:

* NONE–specifies that the signal remain forced unless you:
  1 Place the “free” symbol, a question mark (?), on the last line of the pattern file, or
  2 Use the DEPOSIT/FREE command or
  3 Drive the signal with a DEPOSIT or PATTERN command. NONE is the default.

* ACTIVITY–specifies that the signal keep its forced value until gate activity changes the signal value. For MOS, gate activity refers only to pull-up and pull-down stacks directly connected to the node.

* EQUALITY–specifies that the signal remain forced until gate activity brings its value to one identical to the forced value. For MOS, gate activity refers only to pull-up and pull-down stacks directly connected to the node.

* time–specifies that the signal be free a specified amount of time after the pattern ends.

Primary inputs are not affected by /FINAL_RELEASE.

/NOFINAL_RELEASE
Equivalent to /FINAL_RELEASE=NONE. This is the default.

/FREE_STATE=logic_state
The /FREE_STATE qualifier specifies that its argument, the logic states 1, 0, U#2, Z#2, or ?#2, should be interpreted as the "?" logic state. (The #2 specification must be used with U, Z, and ? states.) The "?" logic state means that a signal should be released from the value that it was forced to
by a DEPOSIT/FORCE command or that it should not be compared when
a PATTERN/COMPARE command is entered.

You can enter the /FREE_STATE qualifier with both ASCII and binary
format pattern files. It is particularly useful with binary format pattern
files because they cannot contain the "?" character.

/LOGNAME=logname
/LOGNAME=(logname,...)

Routes all comment lines in the pattern file that begin with % to a log
file. If a logname has previously been set up with a LOG command, output
is routed to this log file. Otherwise, an implicit SET LOG command is
performed, which opens a log file. The log file remains open until you
explicitly close it with CANCEL LOG. Multiple lognames can be used to
print to multiple files by enclosing the list in parentheses.

/MAP=(original_name=resulting_name,...)

Used with /BINARY, this qualifier translates the signal names in a binary
file, called the original names, to the signal names in another network,
called the resulting names. You can use the /MAP qualifier when some or
all of the signal names in the access list to the TRACE/BINARY command
that created the binary file do not match the signal names in the network
that you are driving with a PATTERN/BINARY command.

The syntax and rules that apply to signal names apply to the original
names and resulting names. A resulting name can also be %NULL.

The wildcard characters (*) and (?) can be used in original names and
resulting names. The rules for mapping original names to resulting names
when wildcard characters are used are as follows:

- A resulting name is matched left-to-right with the wildcard characters
  in an original name.
- Mappings are attempted in the order specified in the argument to the
  /MAP qualifier, and the first successful match is the one used.
- If no match is found, the original name is used.

For example:

PATTERN/BINARY/MAP=(%p=%, *=%NULL)

In this example the values in the binary file that were recorded in traces
of signals whose names ended with the suffix "%p" are scheduled to be
applied to signals that have the same names only without the "%p" suffix.
Values in the binary file that were recorded in traces of signals whose
names did not end in the suffix "%p" are ignored.

The /MAP qualifier can be used to transform vectors into vector or scalar
subsets. In these transformations the following rules apply:

- Signal that are not specified as original names in a /MAP qualifier
  argument have no alterations to their subscripts.
- The entire subscript bit width for an original name is copied to its
  resulting name unless a subscript is included with an original name to
  specify a bit or bit range.
- The wildcard character (*) can be used in subscripts for original names. The asterisk (*) substitutes for a single bit number. For example:

/\MAP:(*%c<=*.)\n
Removes the "%c" suffix, converts the bit numbers to a suffix that begins with an underscore and the bit number.

A signal named xyz%c<7:0> is transformed to a list of signals that are named xyz_7, xyz_6, xyz_5, xyz_4, xyz_3, xyz_2, xyz_1, and xyz_0.

/\MAP:(*%c<:5>=*.)\n
Transforms three of the eight bits in a signal named xyz%c<7:0> to signals named xyz_7, xyz_6, and xyz_5.

/\MAP:(*%c<3:=)*.0\n
Transforms four of the eight bits in a signal named xyz%c<31:0> to signals named xyz_03, xyz_02, xyz_01, and xyz_00.

- The subscript from original name is not automatically copied to the resulting name. If no subscript is specified for a resulting name, it will have no assigned bit number or numbers, but it will have the same bit width as the original name. You can specify a subscript for the resulting name; if you do, the bit width of the resulting name must match the bit width of the original name. For example:

/\MAP:(*<0>=)\n
Removes the <0> from the name.

/\MAP:(*<5>=*.<5)\n
Converts the <5> subscript to a ".<5" suffix.

/\MAP:(*<31:16>=*<15:0>)\n
Changes the bit numbers but not the bit width.

- If the subscripts for an original name specify less than the entire bit range the remaining bits can be used in a subsequent pattern.

- If you specify a subscript for a resulting name without including a subscript for the original name the subscript must be for a bit width of 1.

/\PATTERNNAME= pattern_name

Used in conjunction with the prefixes ENABLE, DISABLE, and CANCEL, Pattern_name is the name of the pattern. If a name is not given in a SET PATTERN command, a name will be created of the form 'PATTERNNnnn', where 'nnnn' is a number that makes this name unique.

/\RADIX= integer

Specifies the radix to be used for the interpretation of vector data. The default radix is base 2 for data, and base 10 for %TIME.

/\RELATIVE

/\ABSOLUTE

Specifies whether the times given in pattern vectors are relative (to the current simulator time) or absolute (with reference to simulation time 0). When /\RELATIVE is used, the pattern vectors may contain absolute time values (specified by use of " I " prefix); such absolute times must appear in ascending order within the vector set. When /\ABSOLUTE is used, all time values in the vector set are assumed to be absolute, regardless of the appearance or absence of the " I " time prefix. Absolute times must appear in ascending order within the vector set. A relative time of 0 or two equal
absolute times can be used to create a “spike.” Time units may not be used with time in a pattern vector file.

/TIMEUNIT=units
The time units for this qualifier are S, MS, US, NS, PS, PS.

The /TIMEUNIT qualifier specifies how DECSIM interprets the time values in the pattern file. This qualifier should be used when you plan to enter a SET TIME/UNITS command after you have entered a PATTERN command but do not intend to change the time units for the time values in the pattern file.

If this qualifier is not entered, DECSIM uses the time unit specified in your last SET TIME/UNITS command; the default time unit for that command is nS. If you do not use either SET TIME/UNITS or /TIMEUNIT, the default is NS.

/TOKEN=’single_char’
Used with /LOGNAME, this qualifier controls whether the %@, which is used to indicate comments in the pattern file that you want included in the log file is included (the default), or whether it is replaced by the single specified character when the comment line is written into the log file. This qualifier affects PATTERN output only. Only %@ can be used to indicate printable comments in the pattern file.

/WIDTH= integer
Specifies the nonzero width of each field in the vector data. This allows fixed-format vector data information.

LOCAL QUALIFIERS
The following qualifiers can be used for individual items in an access list. These qualifiers override the qualifier settings at the beginning of the command. A qualifier on one item does not affect another item’s qualifier value.

/DELIMITER= ‘char’
The ‘char’ is the character that separates one item from another. The default delimiter characters are spaces, tabs, and form feeds.

/RADIX= integer
Specifies the radix to be used for the interpretation of vector data. The default radix is base 2 for data, and base 10 for %TIME.

/WIDTH= integer
Specifies the nonzero width of each field in the vector data. This allows fixed-format vector data information.

DESCRIPTION
The PATTERN command applies input pattern vectors to the simulated network at specified times. You supply these vectors, along with a list of the signal inputs they are to be applied to, in the form of a disk file. This file can be prepared using a text editor or CAD tool, or DECSIM itself (for example, TRACE or WATCH commands).

Note: The output from the TRACE command frequently serves as the input for the PATTERN command.
To override a scalar primary input signal driven by the PATTERN command, you must use DEPOSIT/FORCE. You cannot override gate outputs or MOS nodes driven by the PATTERN command until the end of the pattern file, or until you cancel the pattern file with the CANCEL PATTERN command.

**Note:** Using DEPOSIT/FORCE is similar to "sticking" a signal in fault simulation; remember to use DEPOSIT/FREE when you are through forcing that signal.

PATTERN cannot be used in conjunction with fault simulation to assign a value to an internal node; if it is used, a warning message is issued.

A completed pattern file leaves all driven signals in the state specified by the last line of the PATTERN command. Then, you can deposit to inputs with the DEPOSIT or PATTERN command. Outputs remain forced unless you specified otherwise with the DEPOSIT/FINAL_RELEASE command. You can override forced outputs with \(/FORCE\) or \(/FREE\).

The SHOW PATTERN command displays the names of the currently active patterns, log file names, and associated statistics.

The CANCEL PATTERN command terminates the specified pattern vector set. When this command is given during a simulation time slot (for example, as a WATCH action), all pending transitions for that time slot are canceled, and the /FINAL_RELEASE qualifier takes effect.

The DISABLE PATTERN command allows you to suspend the specified pattern vector set. As a result of this command, no further input transition information is taken from the file until an ENABLE PATTERN command is issued. All pending transmissions are canceled.

The ENABLE PATTERN command reactivates the specified pattern. All transmissions that were not executed because of the DISABLE PATTERN are rescheduled. If the time specified in the current pattern vector has passed, a warning is given and the events are scheduled for the current time.

**Zycad Mode**

In the Zycad environment, the PATTERN command processes the entire pattern and saves only the signal transitions.

A warning is given if two separate patterns stimulate the same signal.

**Note:** If a signal is stimulated by two patterns with overlapping simulation times, the results are unpredictable.

PATTERN can drive a high impedance state (Z#2) only on primary inputs that drive buses.

Just as in soft DECSIM mode, times in a pattern file default to being RELATIVE unless the ABSOLUTE qualifier is given on the PATTERN command.

The CANCEL PATTERN command must be used before the SIMULATE command. You can use it to cancel an entire pattern file; you cannot cancel a portion of a pattern file.
The CANCEL ZYCAD command cancels all patterns read in the ZYCAD environment. Patterns read in the ZYCAD environment are not available during 'soft' simulation, and vice versa. PATTERNS are dynamic objects that are closely tied to the simulation clock, so that active PATTERNS are not easily transported between the 'soft' and 'hard' simulation environments.

The /FINAL_RELEASE qualifier accepts only the "none" and "time" arguments.

Each PATTERN command has a limit of 65K signals.

---

**EXAMPLE**

**PATTERN file:**

The following is the contents of the file MODOUT.PAT. It was generated by a TRACE/VECTOR/LOG command. The first item in each line is the time, followed by a vector of input signals.

```
! MODOUT.PAT
 0 111
 5 110
10 101
15 022
17 022
20 000
.
1000
! End of file MODOUT.PAT
```

**DECSIM commands:**

The following PATTERN command sets up the pattern file MODOUT.PAT as input to the signals IN1, IN2, and IN3. The /WIDTH=1 global qualifier specifies that all the signals are one digit wide. The local qualifier /WIDTH=8 on %TIME overrides the global qualifier /WIDTH=1 for %TIME only.

```
SIM> PATTERN /WIDTH=1 /FILE=MODOUT/ABSOLUTE %TIME/WIDTH=8, IN1, IN2, IN3
%I, [SIMPNI] Pattern name is: PATTERN1
SIM>
SIM>
```
SIM> SHOW PATTERN
File Name: _DBC0:[SHERWOOD]MODOUT.PAT;2
Pattern Name: PATTERN1 Line 5/1 Enabled
SIM> TRACE IN1, IN2, IN3
SIM> SIMULATE 20

0 IN1 U --> 1
0 IN2 U --> 1
0 IN3 U --> 1
5 IN3 1 --> 0
10 IN2 0 --> 1
10 IN3 1 --> 0
15 IN1 1 --> Z
15 IN2 0 --> Z
15 IN3 1 --> Z
17 IN1 Z --> 0
20 IN2 Z --> 0
20 IN3 Z --> 0

Network idle, Simulation Clock: |20 ns
SIM> exit
For more information see:
DEPOSIT TRACE/VECTOR ZYCAD

5-162
PAUSE (Command Macro)

Operates from an indirect file to temporarily suspend input from the indirect file and accept input from your terminal.

**FORMAT**

<table>
<thead>
<tr>
<th>PAUSE</th>
<th>[message]</th>
</tr>
</thead>
</table>

**Default Qualifiers**

None.

**Optional Qualifiers**

None.

**restrictions**

- This command can only be executed from an indirect file.
- In batch jobs DECSIM executes no commands that follow the PAUSE command macro in an indirect file.

**DESCRIPTION**

The PAUSE command macro executes a DISABLE INDIRECT to temporarily get input from the user terminal. The CONTINUE command macro resumes indirect file processing by using an ENABLE INDIRECT command. If a message is included as an argument to the PAUSE command macro, it is printed on your terminal.

For more information see:

- CONTINUE (a command macro)
- ENABLE/DISABLE INDIRECT

**EXAMPLES**

SIM> PAUSE after initialization power up sequence

SIM> PAUSE 'read command file "test1.ind;3"'

**PAUSE Macro Definition**

SET MACRO /COMMAND /ABBREV=3 PAUSE [ _$TEXT_ ] =
  print ",", "$TEXT"; disable indirect;
SENDMACRO
PRINT

Allows you to display formatted text and values on the terminal or, via the DECSIM LOG command, to write this information to files.

FORMAT

PRINT [/global_qualifier...] [arguments,...]

Default Qualifiers
/NOZEROFILL
/RADIX=10
/TTY

Optional Global Qualifiers
/ALL
/LOGNAME=logname
/LOGNAME=(logname,...)
/NOCRLF
/RADIX=integer
/RESETTIME

Optional State Argument Qualifiers
/TIMEUNIT=timeunit
/WIDTH=integer

Optional State Display Qualifiers
/0='x'
/1='x'
/Z='x'
/U='x'

restrictions
None.

ARGUMENTS

[arguments,...]
Arguments may be any of the following:

'any text'
variables [ /local_qualifiers ]
signal names or states
( expression ) [ /local_qualifiers ]
&system_variable [ /local_qualifiers ]

Arguments may be intermixed but must be separated by commas. Expressions must be enclosed in parentheses.
The following list describes the most commonly used print system variables.

These variables can also be used as operands in expressions. If they are used as a part of an expression, the expression in the PRINT command must be in parentheses. For example, PRINT (%TIME + 1).

For a complete list of system variables, see Section 4.2.6.3.

%CRLF
Outputs a CRLF where %CRLF occurs, unless /NOCRFLF is specified. This is in addition to the CRLF at the end of a PRINT statement.

%DATER
Types the date, for example, “2–Dec–1984”.

%DAYTIME
Types the time of day, for example, “13:10:01”

%DELTATIME [/time qualifier ]
Outputs the simulated clock time since the time of the last PRINT/RESEETTIME statement to this LOG file, in the current units or the /TIMEUNITS specified.

%ERASE_LINE
Clears the remainder of the line after the cursor.

%ERASE_SCREEN
Clears the remainder of the screen after the cursor.

%FILESPEC
Types the complete file specification of the target file as set in the LOG command. Prints each file’s name with /ALL.

%HOME
Moves the cursor to the home position (row 1, column 1).

%INFILE(file)
Types text found in file.

%POSITION (row, column)
Moves the cursor to the specified position. Row should be a value or an expression that evaluates in the range of 1 to 24. Column should be in the range of 1 to the width of the screen.

%REVISION
Types the network revision number.

%TIME [/time qualifier ...]
Types the absolute simulator clock time on the output line in the current units specified by the SET TIME/UNIT command, or by the PRINT qualifier /TIMEUNIT.

%TITLE
Types title (if any) given in LOG command.
**QUALIFIERS**

The PRINT command has global qualifiers and qualifiers for state arguments and time value arguments.

Zero or more of the following qualifiers are allowed per PRINT command:

/0='x'
/1='x'
/Z='x'
/U='x'

Use the string 'x' wherever a binary 0, 1, Z, or U value is to be printed. This is useful for graph-type traces and other representations for Z and U.

/ALL
Prints to all output files, but not to the TTY= terminal.

/LOGNAME=logname
/LOGNAME=(logname,...)
Routes output (from this PRINT command only) to a log file. If a logname has been previously set up with a LOG command, output is routed to this log file. Otherwise, an implicit SET LOG command is performed, which opens a log file. The log file remains open until you explicitly close it with CANCEL LOG. Multiple lognames can be used to print to multiple files by enclosing the list in parentheses.

The Table 5-4 shows the effect of a [NO]LOGNAME qualifier on the PRINT command.

<table>
<thead>
<tr>
<th>Table 5-4</th>
<th>Routing of output from PRINT/[NO]LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINT 'ABC'</td>
<td>terminal yes</td>
</tr>
<tr>
<td>PRINT/NOLOG 'ABC'</td>
<td>yes</td>
</tr>
<tr>
<td>PRINT/LOG=logone 'ABC'</td>
<td>yes</td>
</tr>
<tr>
<td>PRINT/LOG=logtwo 'ABC'</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Key**

yes—the character string 'ABC' is printed here
logone—defined as /RESPONSE/NOCOMMAND
logtwo—defined as /NORESPONSE/NOCOMMAND

/NOCLRlf
Inhibits automatic (default) carriage return/linefeed at end of the PRINT statement

/RADIX=integer
Prints values in RADIX integer. Default radix is that set by SET RADIX command.
/RESETTIME
Resets time counter for /DELTATIME reports for the particular LOG file being written to.

/TIMEUNIT=timeunit
Changes the reporting time units for this PRINT command or time variable.

/TTY
Routes printing to the terminal. Can be used in conjunction with /LOGNAME or /ALL to route printing to log files and the terminal.

/WIDTH=integer
Expands PRINT field argument to fit in a given character width field (not bit width) by adding spaces (or zeroes if /ZEROFILL is selected) on the left-hand side of the value field. If the value to be printed requires a field larger than the /WIDTH specified, the needed width of the field (that is, complete value) is printed with no truncation. This qualifier does not affect the width of literal character strings.

/ZEROFILL
/NOZEROFILL (D)
Fills enough zeroes to satisfy the accompanying /WIDTH qualifier field width. /WIDTH must be given in conjunction with /ZEROFILL.

DESCRIPTION
The PRINT command allows you to annotate TRACE output and print formatted information to files and to the terminal. TTY (TT) refers to your terminal. When you submit a batch job, the batch .LOG file records the terminal output.

Quoted arguments (in single quoted text strings) with PRINT are written directly into the file or to the terminal without quotation marks. The text string follows the rules for quoted-strings under "STRINGS".

If the same arguments are used for several PRINT commands, then a text string macro should be defined and used.

To print a single quotation mark, type it twice. Thus, PRINT 'abc"def' will give abc'def as output. To print a CTRL/x character, use the up arrow (^) followed by the desired x character. Two up arrows enclosed in single quotation marks print up arrow (^). For a CTRL/^ type "^~" or "^>".

Note: Break the print (especially if a long PRINT statement is in a macro definition) into several statements (not just line continuations) to avoid line length restrictions. Use /NOCRLF to make the output appear to be one print statement.

For more information see:

LOG SET TIME WATCH

5-167
EXAMPLES

SIM> PRINT 'Value of A = ', A<15:0>/RADIX=8, ' in octal'
Value of A = 105704 in octal
SIM> STATE ir<15:0>

SIM> macro decode=
MACRO> if ir<14:12> eqi 0 then ! single operand instr
MACRO> begin
MACRO> PRI/nocrlf 'SINGLE OPERAND INSTRUCTION', %crlf, 'I', ir/rad=8, ' = '
MACRO> select ir<11:6> from
MACRO> [50#8] : PRI/nocrlf 'CLR';
MACRO> [51#8] : PRI/nocrlf 'COM';
MACRO> [52#8] : PRI/nocrlf 'INC';
MACRO> [53#8] : PRI/nocrlf 'DEC';
MACRO> etc ...
MACRO> [otherwise]: PRI/nocrlf '???';
MACRO> endselect;
MACRO> end

MACRO> if ir<15> eqi 1 then PRI/nocrlf 'B';
MACRO> PRI ' R', ir<2:0>/rad=8
MACRO> end
MACRO> else
MACRO> begin
MACRO> PRI/nocrlf 'DOUBLE OPERAND INSTRUCTION', %crlf, 'I', ir/rad=8, ' = '
MACRO> select ir<14:12> from
MACRO> [1] : PRI/nocrlf 'MOV';
MACRO> [2] : PRI/nocrlf 'CMP';
MACRO> ! etc,
MACRO> endselect;
MACRO> end

MACRO> if ir<15> eqi 1 then PRI/nocrlf 'B';
MACRO> PRI /rad=8 ' R', ir<8:6>, ' R', ir<2:0>
MACRO> end
MACRO> $endmacro

SIM> ev ir=105104#8
SIM> decode
SINGLE OPERAND INSTRUCTION
105104 = COMB R4
SIM> ev ir=020304#8
SIM> decode
DOUBLE OPERAND INSTRUCTION
20304 = CMP R3, R4
SIM>
Sets the number base used to display logic values and numeric values.

**FORMAT**

[SET] RADIX  decimal_integer
CANCEL RADIX
SHOW RADIX

**Default Qualifiers**

*None.*

**Optional Qualifiers**

*None.*

**restrictions**

- The RADIX command does not affect the display of time values, logic values, or typed-in numeric values.
- The decimal integer must be no less than 2 and no greater than 16.

**Initialization default**

SET RADIX 10

**DESCRIPTION**

The RADIX command sets up a default radix that is used when logic values and numeric values are displayed by commands such as EVALUATE, PRINT, and EXAMINE.

The RADIX command does not affect the display of time values, which are always displayed in base 10.

The radix of a logic value or numeric value being typed in is not governed by the SET RADIX command; it is governed only by the radix indicated on the value (with the pound sign). The default type-in radix is 10.

The current RADIX setting is stored in an internal list as a stack such that the previous setting is stored. Up to ten settings can be stacked. CANCEL RADIX returns the RADIX setting to the previous setting. This allows a MACRO definition or WATCH action to set the radix and then return the RADIX setting to the previous value.

SHOW RADIX displays the current RADIX setting.

For more information see:

DEPOSIT  PATTERN  WATCH
EXAMINE  PRINT

Section 4.2.3 and Section 4.2.4.
RECALL

Displays previously entered commands so that they can be reprocessed.

FORMAT

RECALL/ALL
RECALL integer
RECALL command_specifier

Default Qualifiers
None.

Optional Qualifiers
/ALL

restrictions
None.

ARGUMENTS

command_specifier
A command specifier is the command keyword that RECALL will search the list for.

QUALIFIERS

/ALL
Specifies that you want to see a display of your last 20 command lines. DECSIM displays a numbered list of your last 20 command lines beginning with the command line you entered before your RECALL/ALL command.

DESCRIPTION

The RECALL command is used to display previously entered commands so that they can be reprocessed.

You can use RECALL in three ways. RECALL/ALL recalls the last 20 command lines. Each command in the list is preceded by a number. RECALL integer recalls the command whose number is specified. RECALL command_specifier recalls the command whose keyword or abbreviation is specified. Only the most recent match will be found.

The RECALL command itself is not added to the recall list. RECALL works in ZYCAD mode.
EXAMPLES

SIM> recall/all
1) sym
2) sim 10
3) sh ti
4) trace/odo sig
5) sim 50
6) sym
7) sim 50
8) dep din = [1:100, 0:100]8
9) dep enable = 1
10) dep set = [1:400, 0:400]2
11) dep cir = [1:800, 0:800]1
12) sym
13) @test1
14) watch/every=400 do print/logname=table '------------'
15) log/nocommands/noresponses/noecho/table
16) dep clk = [0, [1:50, 0:50]16]
17) sym
18) use t latch
19) log/commands/responses/echo record
SIM> recall 5
SIM> sim 50
SIM> recall deposit
SIM> dep clk = [0, [1:50, 0:50]16]

This example shows all three ways of using the RECALL command.

SIM> pattern/file=adder/radix=10/absolute/patternname=exp
MORE> %time/width=2, a1, b1, a2, b2, c0
%I-PNI, The pattern name is EXP
SIM> cancel pattern exp
%I-CAP, Canceling the pattern EXP
SIM> recall pattern
SIM> pattern/file=adder/radix=10/absolute/patternname=exp

This example shows a PATTERN command that used more than one line. When the RECALL PATTERN command was entered only the first line of the PATTERN command was displayed.
REPLAY

Reads the binary file generated with the TRACE/REPLAY or TRACE/BINARY command so that DECSIM can display the traced data from a previous simulation. You can examine this data as you would during a simulation.

**FORMAT**

REPLAY/FILE = file
REPLAY [ | ] time

Default Qualifiers
None.

Optional Qualifiers
/FILE=file_name

**restrictions**

- REPLAY shows the results of a previous simulation; it does not, however, resimulate, so you cannot use the SIMULATE command while replaying a run.
- REPLAY ignores traces of Behavior or Memory nodes.
- REPLAY does not report events that are scheduled to occur.
- REPLAY cannot be used for fault simulation runs.

**ARGUMENTS**

*time*
Time is an integer that can be followed by a time unit specification.

**QUALIFIERS**

/FILE=filename
The /FILE qualifier is entered to read into DECSIM the trace information stored in a .TRA file that was generated by the TRACE/REPLAYABLE/LOG or TRACE/BINARY/VECTOR/DELAYED/LOG command (see the TRACE command for information on /REPLAYABLE).

**DESCRIPTION**

To recheck a previous simulation, follow this procedure:

1. Enter DECSIM.
2. Enter the REPLAY/FILE=filename command.
3. Enter the REPLAY command followed by a time to set the clock. In this way, you can move backwards and forwards in time.
4. Enter the EXAMINE command to check signals at the specified times.
5. Enter the USE or the RESTORE command to leave REPLAY mode and reenable DECSIM's simulation capability.
EXAMPLE

SIM> use adder
    Reading File: DOC$DISK:[GRENNE.DECSIMCLASS]ADDER.NOB;11
%I-TRS, Time Resolution = 1 nS, Default unit is nS
SIM> replay/file=tracing
%I-RFL, REPLAY file successfully loaded.
SIM> replay 40
SIM> sym
   A1 = 0
   A2 = 0
   B1 = 0
   B2 = 0
   C2 = 0
   CO = 0
   SUM1 = 0
   SUM2 = 0
SIM> replay 80
%I-TNF, Specified time not found, using closest previous time instead.
    Simulation Clock: |40 nS
SIM> replay 100
    Simulation Clock: |100 nS
SIM> sym
   A1 = 1
   A2 = 1
   B1 = 1
   B2 = 1
   C2 = 0
   CO = 1
   SUM1 = 0
   SUM2 = 0
SIM> labels
   LEVEL_D: Model = ADDER Revision = 0
SIM> scope level d
    Scope = LEVEL_D, Model = ADDER, Revision = 0
SIM> sym
   LEVEL_C_1.A1 = 1
   LEVEL_C_3.A2 = 1
   LEVEL_C_1.B1 = 1
   LEVEL_C_3.B2 = 1
   C2 = 0
   LEVEL_C_1.CO = 1
   OUT6 = 1
   SUM1 = 0
   SUM2 = 0
SIM> e/fanin/fanout out6
   OUT6 = 1
   OUT6 ===> LEVEL_C_1.OUT6, LEVEL_C_3.OUT6, LEVEL_C_4.OUT6
   OUT6 <= LEVEL_C_2.OUT1 = 0, LEVEL_C_2.OUT2 = 0, LEVEL_C_2.OUT3 = 0

5-174
**RESTORE**

Reloads a network and its state from a disk file created by the SAVE command.

**FORMAT**

```
RESTORE filename
```

*Default Qualifiers*

`/CHECK`

*Optional Qualifiers*

`/NOCHECK`

**restrictions**

- You must save and restore a particular network under the same version of DECSIM.
- You must save and restore a particular network under the same version of VMS.

**QUALIFIERS**

`/CHECK`

`/NOCHECK`

The `/NOCHECK` option shuts off the VMS version check. The DECSIM maintainers do not guarantee that the network will be properly restored if you use this option.

**DESCRIPTION**

The RESTORE command reinstates a network and its state. Where multiple states of the network are saved (with the SAVE/EVERY command or multiple SAVE commands), simulation can be restarted from any state by specifying the appropriate file specification in the RESTORE command.

For more information see:

SAVE
RESUME (Command Macro)

Terminates the current interactive debug session and returns to simulation at the point at which it was interrupted. RESUME is functionally equivalent to the CANCEL BREAK command.

FORMAT

RESUME

Default Qualifiers
None.

Optional Qualifiers
None.

restrictions

• The RESUME command macro must be executed interactively.

DESCRIPTION

For more information see:

CANCEL BREAK SET BREAK
DISABLE BREAK SHOW BREAK
ENABLE BREAK STEP

EXAMPLE

SIM> WATCH /CALL=rtna DO BREAK
%I, [SIMWTN] WATCH/TRACE Name is WATCH1
SIM> CALL rtna
BREAK from WATCH /WATCHNAME:WATCH1 /CALL:RTNA
SIMPBREAK>
...
SIMPBREAK> RESUME
SIM>
SAVE

Allows you to save the state of the network at any time and restore that state later.

FORMAT

[SET] SAVE [/qualifier... ] [ file ]
CANCEL SAVE

Default Qualifiers

NOCOMPRESS

Optional Qualifiers

/AT:time
/COMPRESS
/EVERY=[ ]time

restrictions

None.

ARGUMENTS

file

Specifies the file to use to save the network state. The default extension is .NSV (for Network Save file).

QUALIFIERS

/AT:time

Automatically saves the network state at the specified time. If relative time is given, then the save is done at the current time added to the specified time. This is a one-time save. If the file extension is not specified, the default extension is assumed.

/COMPRESS

NOCOMPRESS (D)

/COMPRESS uses the data compression facility to write a compressed file. When you use the /COMPRESS qualifier DECSIM may take more time to save and restore the state of your network but the .NSV file takes less disk space. /NOCOMPRESS writes the .NSV file in an uncompressed format. It is the default.

/EVERY=[ ] time

Automatically saves network every "time" units of time during simulation. The SAVE/EVERY command is executed at the end of the simulation time slot. If a time (absolute time) is given, a one-time SAVE is executed at the specified absolute time.

Note: With /EVERY, the file extension is ignored in the command line. Instead, a fixed extension is provided beginning with "000" and incrementing with each /EVERY save.
DESCRIPTION

The SAVE command writes a file containing the entire network and state information. You can reinstate this network with RESTORE.

SAVE facilitates checkpointing long simulation runs so that fragments of the simulation can be repeated from a clean state.

When SAVE is executed, it saves a file and continues. When the file is restored, I/O and command state are back to a clean simulation top level. When you restore the file, you sometimes have to make adjustments. For example, if a SAVE command were issued in the middle of a FOR loop, the FOR loop continues and operates normally—the SAVE has no effect on the simulation or command sequence. However, when that particular saved network is restored, the FOR loop will have been lost, and you are in top-level command mode. If you have a pattern file in progress when you execute a SAVE command, you must restart your pattern file from where it left off when you restore the file.

The default file specification is SIM concatenated with the process ID number. The file extension is .NSV. For example, if your process number were 0215 0123, then the file name would be SIM215123.NSV.

The SAVE command saves the following:

- Static Network Information: Symbols, labels, states, models, zoom table, parameter table and interconnection, and network mapping variables.

- Dynamic Network Information: Current states, time wheel, scheduled transitions, time, fault sources and effects.

- Command Information: Command states and macros. The commands SCOPE, RADIX, and CONFIGURE are saved.

Every version is saved with the SAVE/EVERY command. Use the DCL SET FILE/VERSION_LIMIT=integer filename command to limit the number of versions saved.

The CANCEL SAVE command cancels the currently active SAVE. It applies only to the automatic (qualified) SAVE command. Use this command to stop a previous SAVE with the /EVERY qualifier.

For more information see:

RESTORE
Scope

Moves your observation point up or down the network hierarchy, simplifying
the observation and reporting of signals, memories, and variables within
particular hierarchical entities (such as a model element).

Format

[SET] SCOPE [/qualifier] scope_name
CANCEL SCOPE [/qualifier]
SHOW SCOPE [/qualifier]

Default Qualifiers
None.

Optional Qualifiers
/ALL
/QUIET

Restrictions
- If the SCOPE stack is full (8 scope settings) when you execute the
  SCOPE command, the earliest scope setting will be deleted.

Arguments

scope_name
Either "%NET" (which indicates top-level scope) or a sequence of model
instance labels, separated by periods.

Qualifiers

/QUIET
[SET] SCOPE/QUIET disables the informational message that displays
the current scope setting.

CANCEL SCOPE/QUIET disables the informational message that displays
the canceled scope.

This qualifier is invalid in SHOW SCOPE commands.

/ALL
SHOW SCOPE shows the current SCOPE setting. SHOW SCOPE/ALL
shows all SCOPE settings in the stack.

CANCEL SCOPE/ALL cancels all scope settings in the stack and returns
the user to the %NET level.

This qualifier is invalid in SET SCOPE commands.
DESCRIPTION

The SCOPE command simplifies the accessing of signal names in the portion of the circuit you're debugging. The command sets up a window on signal/variable accesses. The argument "scope name" specifies beginning hierarchical name components for all signals. Each hierarchical level (as specified by the network description element instance label) is separated by a dot (.) Normal signal name rules apply to what is next to the dot; leading or trailing spaces and tabs are ignored.

This can be compared to specifying a mail address. You could specify "SET SCOPE Mass.Hudson.ReedRd.77." This would set the scope to 77 Reed Road in Hudson, Mass. You could then specify signal names (specific rooms in the building) without specifying the hierarchy in which they are located.

The SCOPE commands allows you to access both signals at the top level of hierarchy and at the level to which you have set scope without typing full hierarchical names. To access signals outside the context of the SCOPE command, you must type the full hierarchical name.

You can abbreviate levels of hierarchy as long as the abbreviations are not ambiguous.

The SCOPE command affects the accessing of signals from the command language, particularly from EXAMINE and WATCH commands.

It is possible to give up to six SET SCOPE commands without giving any corresponding CANCEL SCOPE. This is an implementation restriction. It is possible to set scope to several locations at once.

%NET is a special scope name. It allows you to return to the top-level SCOPE. In this way you can examine a top level signal with a name that is the same as a signal in the current SCOPE.

DECSIM always looks for a signal in the current scope first. If the signal is not found in the current scope, then it tries to find it at the top (%NET) level. If the signal is not in the defined scope or the top level, an error message is issued.

The SCOPE command does not affect accessing command states; it only affects network variables.

For more information see:

DEPOSIT EXAMINE
EVALUATE USE
EXAMPLES

SIM> USE tstsco
Reading File: DBC1:[PHILLIPS] TSTSICO.MOB;1
Time Resolution = 100PS (default units are NSecs)
SIM> SHOW SCOPE
  Scope = %NET, Model = Network, Revision = 0
SIM> LABELS
  E1: Model = CHIP_II, Revision = 0
  E2: Model = CHIP_FF, Revision = 0
  E3: Model = CHIP_IF, Revision = 0
  E4: Model = CHIP_FF, Revision = 0
SIM> SET SCOPE E1.C2
SCOPE = E1.C2, Model = CELL, Revision = 0
SIM> CANCEL SCOPE
SCOPE = %NET, Model = Network, Revision = 0
SELECT

Chooses 0, 1, or more statements or blocks to execute. A trigger expression determines whether or not the associated statement is to be executed. The SELECT command is similar to the SELECTALL statement in the behavioral modeling language.

FORMAT

[optional qualifiers] SELECT (test_condition) FROM
[trigger_expr_1]: stmt_or_block;
.
.
[trigger_expr_n]: stmt_or_block;
ENDSELECT

Default Qualifiers
None.

Optional Qualifiers
None.

restrictions

- You may not abbreviate the keywords: SELECT, FROM, ALWAYS, OTHERWISE, and ENDSELECT.
- No trigger expression and statement can follow the [OTHERWISE] case and its associated statement.
- The trigger expression can be a list of values, for example [1, 3, 5], but unlike the SELECTALL statement in the behavior language, it cannot be a range of values.

DESCRIPTION

The SELECT control command prefix allows complex conditional execution of a list of DECSIM commands. The condition triggering the execution of the list of commands is whether or not the evaluated trigger expression is the same as the SELECT test_condition. DECSIM compares the two and if their values match, the associated command is executed; if they don’t, the command is ignored.

Evaluation of trigger expressions and conditional execution of commands proceeds sequentially. It begins with the first trigger expression, and continues in order.

A trigger expression may actually be a list of expressions. In this case, the first true expression causes execution of the associated command. More than one true trigger expression in a single list has no special effect – the associated command is still executed only once.
Trigger expressions may contain wildcard constants (that is, constants with "?” digits in them). Note that only radix 2,4,8, and 16 constants may have "?” wildcards in them.

Two special keywords that may be used in place of a trigger expression or expression list are OTHERWISE and ALWAYS. These may not be mixed with expressions in a single expression list. OTHERWISE produces a successful compare (and execution of its associated command) only if no other trigger expression was previously true. ALWAYS forces execution unconditionally on its associated command. Once an ALWAYS command has been executed, no subsequent OTHERWISE command can be successful.

A command can be any DECSIM command, including another SELECT or other iterative control command. Several commands may be grouped together to be treated as one by enclosing them in a BEGIN–END block.

It is recommended, but not necessary, to format a SELECT command as shown above in the format section. The format shows semicolons ending the trigger commands. Actually, semicolons do not end the command. A semicolon is not necessary, but is recommended.

You may be able to use the "?” wildcard character to work around the prohibition in value ranges in trigger expressions. For example, you cannot enter [0:7], but you can enter [??2], which means any value up to three bits wide.

Control flow tracing can be turned on with the DEBUG/LOOP command.

LEAVE commands may be used from within a SELECT command.

For more information see:

IF

LEAVE

EXAMPLES

Control Flow Tracing

The following tracing is shown when the "DEBUG / LOOP" command is in effect.

SIM> SELECT 3 FROM
MORE> [??2]: PRINT '??2'
MORE> [0]: PRINT '0'
MORE> [0??2]: PRINT '0??2'
MORE> [ALWAYS]: PRINT 'ALWAYS'
MORE> ENDSIM

[ SELECT expression = 3 ]
[ Trigger Expression 1 on SELECT line 1 MATCHES ]

??2
[ Trigger Expression 1 on SELECT line 2 DOES NOT MATCH ]
[ Trigger Expression 1 on SELECT line 3 MATCHES ]

0??2
[ SELECT line 4 ALWAYS being executed ]
ALWAYS
SIM>
SHARE

Enables an external applications program to access DECSIM's data structures during simulation by declaring a portion of the DECSIM address space as a global section. Using the VMS event flag services, the two processes can synchronize access to the simulation data structures.

One of the main uses of the SHARE command is to enable the VAX Architectural Exerciser (AXE) to drive a DECSIM model and determine if it meets VAX architectural requirements. For more details, see the /AXE qualifier below.

On a more limited scale, the /MAILBOX qualifier enables an external applications program to interrupt the DECSIM process and send a command to DECSIM to execute.

**FORMAT**

```
SHARE  [/qualifier... ] access_list
```

**Default Qualifiers**

None.

**Optional Qualifiers**

/AXE
/MAILBOX=(COMMAND [,PROMPT])
/NAME

**restrictions**

- The system parameter GBLPAGES must be larger than the network size. See STATISTICS/NETWORK for more information.
- You cannot TRACE behavioral states while a network is being shared. TRACE changes the memory protection, which is not compatible with shared global sections.

**ARGUMENTS**

```
access_list
```

A list of signals or states for which descriptors are generated at the beginning of the global section.

**QUALIFIERS**

/AXE

Specifies that the shared memory interface is specific to the AXE (VAX Architectural Exerciser) application only. Creates a global section with the name "AXEnnnnmm", where nnnnnnm is the user's UIC.

The process of driving a model with AXE can be divided into three steps. In the first step, Axe initializes the global section. In the second step, the model copies the data from the global section to its own registers and simulation begins. After simulation stops, the model copies the new data from its registers back to the global section. In the third step, AXE copies data for the next simulation into the global section or signals that there will be no further simulation.
The following table describes how the VMS event flags coordinate the access of AXE and the model to the global section.

<table>
<thead>
<tr>
<th>Step</th>
<th>DECSIM</th>
<th>AXE</th>
</tr>
</thead>
<tbody>
<tr>
<td>step 1</td>
<td>waits for axe_ready_flag</td>
<td>initializes global section, sets axe_ready_flag</td>
</tr>
<tr>
<td>step 2</td>
<td>clears axe_ready_flag,</td>
<td>waits for model_ready_flag</td>
</tr>
<tr>
<td></td>
<td>copies data, runs model,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>copies data back, and sets</td>
<td></td>
</tr>
<tr>
<td></td>
<td>model_ready_flag</td>
<td></td>
</tr>
<tr>
<td>step 3</td>
<td>waits for axe_ready_flag,</td>
<td>clears model_ready_flag, sets up data for next case and sets axe_ready_flag</td>
</tr>
<tr>
<td></td>
<td>but stops if stop_axe_flag</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is set</td>
<td>or sets stop_axe_flag</td>
</tr>
</tbody>
</table>

The two processes cycle through steps 2 and 3 until AXE sets the stop_axe_flag and simulation stops.

In order to drive a DECSIM model with AXE, the model must have a behavioral module that interfaces it with AXE. See the Examples section below for a template of such a module.

/MAILBOX = COMMAND [,PROMPT]

Creates a command mailbox through which another application can communicate with and control the DECSIM process. Whenever the other process sends a DECSIM command to the mailbox, DECSIM interrupts its current activity and executes the command. The name of the mailbox is the same name as the global section would have been, with _MBX appended to it.

Note: The /MAILBOX switch creates only mailboxes; it does not create a global section.

Because DECSIM may not respond correctly if one command is interrupted to execute another, the DECSIM process should not be executing its own command stream when the other process writes to the mailbox. Also, unless the prompt mailbox is used, the other process has no way of knowing if the command is properly executed. Only command sequences that are guaranteed to be correct should be used without the keyword prompt.

The prompt keyword in this qualifier creates another mailbox, with MBX_PROMPT appended to it. After the other process submits a command, DECSIM writes its prompt to this mailbox. If the command was accepted and executed, then a SIM> prompt is returned. If the command was incomplete, then a MORE> prompt is returned. Use the prompt mailbox to synchronize and verify command execution.

/NAME = name

Specifies the name of the global section (and section file), overriding the default of AXE<nnnnnmm or SHR<nnnnnmm, where nnn is the user's three-digit group number and mmm is the three-digit member number. Users who have more than three digits in either the group or member field
may find it necessary to use the /NAME qualifier to avoid generating an "%E-WLD, invalid wildcard operation" error.

Although the /NAME qualifier does allow you to run more than one SHAREd process on the same system, you should be aware that large networks require proportionally large global page tables and section files for paging. In a heavily loaded system, there may not be sufficient resources to start both DECSIM and the other applications, or the response time may deteriorate.

**Note:** When naming your global section, keep in mind that the global section is a *group* global section, meaning that it can be shared only by processes executing with the same group number.

### DESCRIPTION

To write an application program that accesses DECSIM's data structures during simulation, you need to know the format of the global section and how VMS event flag services coordinate the activities of DECSIM and the applications program. The following sections explain the global section and VMS event flag services. The final section describes specific requirements of the applications program and the model.

**The Global Section**

The SHARE command creates a global section made up of a Header portion and a data portion. The data portion contains all the network data structures. The Header portion contains offsets from the start of the Header to these data structures. The Header also contains information about the signals being stored.

**Figure 5-1 Layout of the Global Section**

```
DECSIM address space

SECTION HEADER

network data structures and
other related data

"SHRnnnnmm" global section
```
The Section Header acts as an index to the data structures. The fixed part of the Header contains offsets to all network structures, and the variable part contains offsets to the fixed topology for the signals and states being shared. Figure 5–2 shows the format of the Header.

**Figure 5–2 Format of the Global Section Header**

<table>
<thead>
<tr>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of entries in header</td>
</tr>
<tr>
<td>total size of header</td>
</tr>
<tr>
<td>number of signals/states in list</td>
</tr>
<tr>
<td>address of workspace in DECSIM</td>
</tr>
<tr>
<td>offset to string pool</td>
</tr>
<tr>
<td>offset to STBV</td>
</tr>
<tr>
<td>offset to label tree</td>
</tr>
<tr>
<td>offset to zoom table</td>
</tr>
<tr>
<td>offset to fixed topology</td>
</tr>
<tr>
<td>offset to dynamic topology</td>
</tr>
<tr>
<td>offset to parameter table</td>
</tr>
<tr>
<td>offset to signal name table</td>
</tr>
<tr>
<td>offset to fixed topology for signal/state</td>
</tr>
<tr>
<td>offset to fixed topology for signal/state</td>
</tr>
<tr>
<td>offset to fixed topology for signal/state</td>
</tr>
</tbody>
</table>

---

**Synchronizing the Two Processes**

VMS event flag services synchronize the access of DECSIM and the other application to the network data. The application and the model must both have routines that read, set, clear, and wait for the shared events flags. For example, the following is a behavioral routine in a DECSIM model that puts the model in a wait state while the other application is running.

```c
ROUTINE MAIN ();
    %SET EVENT_FLAG(70);       !tells other application to start
    %WAIT FOR EVENT_FLAG(71);  !waits until other process is done
    EXECUTE_MODEL();           !calls another routine
ENDROUTINE MAIN;
```

5–187
The corresponding routine in a BLISS application would look like this:

```
ROUTINE MAIN ();
  WAIT FOR EF(70);       !waits until model is ready
  INIT_GLB_SEC();        !initializes global section
  SET EF(71);            !tells model to run
ENDROUTINE MAIN;
```

Application and Model Requirements

If you want to write your own application to run in conjunction with a DECSIM model, keep in mind the following requirements:

1. **The application must map the global section into its address space.**

   If the application is written in BLISS, you can link it with the precompiled BLISS module SYSCALL, which contains the MAP_GLOBAL_SECTION routine. To obtain this module, contact DECSIM::DECSIM_SUPPORT.

   This routine requires an argument, the address of a 2–longword vector, to return the start and end address (bounds) of the global section. The routine itself returns a value, 1 or 0, depending on the success or failure of the mapping. In the case of failure, a message describing the cause of failure is displayed.

   This routine also assigns the VMS event flags 64 to 95 to the application.

2. **The application may have to transform the pointers in the network data structures.**

   If the SHAREd signals or states are four–state or wider than 32 bits, the network data structures contain pointers to other data structures, not the data itself. These pointers are valid in the DECSIM address space only, not in the address space of the application. To retrieve the data itself, the application must transform the pointers, as follows:

   ```
   actual_addr = (.pointer - .addr_of_workspace_in_DECSIM) + .section_addr_low + .total_size_of_header
   ```

   The variable `addr_of_workspace_in_DECSIM` refers to the address of the start of data structures in the DECSIM address space. The variable `total_size_of_header` gives the length of the header. These values are available in the section header.

3. **The application must define the fields in the various network data structures.**

   If the application is written in BLISS, the following REQUIRE files can be included in the application program to define the fields accessed by their names:

   - HDRFLD.REQ—defines the fields in the Section Header
   - SIMFLD.REQ—defines the fields in the network data structures
   - BEHFLD.REQ—defines the fields in the behavioral data structures

   To obtain these files, contact DECSIM::DECSIM_SUPPORT.

4. **The application must include a routine that returns the VAX/VMS user identification code fullword for the current account.**
If the application is written in BLISS, you can link it with the precompiled BLISS module SYSCALL, which contains the routine GET_UIC.

5. The application must include routines that set, clear, read, and wait for VMS common event flags.

If the application is written in BLISS, you can link it with the precompiled BLISS module SYSCALL, which contains the following event flag routines:

- READ_EF()
- WAIT_FOR_EF()
- SET_EF()
- CLEAR_EF()

The READ_EF routine returns the current 1-bit value of the specified flag. It accepts two arguments, the flag number and the address of a variable to return the value of the event flag. This is the only routine that requires two arguments.

The remaining routines require just one argument, the event flag number. The WAIT_FOR_EF routine tests the specified event flag and returns immediately if the flag is set. Otherwise, the process is placed in a wait state until the event flag is set. The SET_EF routine sets the specified event flag to 1. The process waiting for the event flag now can run. The CLEAR_EF routine sets the specified event flag to 0.

All these routines return a value, 1 or 0, depending on the success or failure of the requested service.

6. The DECSIM model must contain routines that set, clear, read, and wait for VMS common event flags.

The behavioral language has the following synchronization primitives, which call the VMS event flag services:

- %READ_EVENT_FLAG(flag_number)
- %WAIT_FOR_EVENT_FLAG(flag_number)
- %SET_EVENT_FLAG(flag_number)
- %CLEAR_EVENT_FLAG(flag_number)

The %READ_EVENT_FLAG primitive accepts only one argument, the event flag number, and returns the value of the flag as the value of the routine. The routine READ_EF, on the other hand, accepts two arguments, as described above. This is because the behavior language primitives return on success only. In the case of failure, an error message is displayed on SYS$OUTPUT and the condition is SIGNALed with the UNWIND option.
EXAMPLE

! This module is a template of the main routines used to
! interface a VAX model with the AXE program

MODULE axe;
BEHAVIOR;

CONSTANT
! event flag numbering convention, determined by AXE
  axe_ready_flag = 64, ! AXE has set up the case
  model_ready_flag = 65, ! Model has executed case
  stop_axe_flag = 66; ! AXE wants to stop the model

ROUTINE setup_before_axe_case<>;
  IF %READ_EVENT_FLAG(stop_axe_flag) THEN RETURN 1;
  %WAIT_FOR_EVENT_FLAG(axe_ready_flag); ! Synchronize with
  %CLEAR_EVENT_FLAG(axe_ready_flag); ! AXE setup
  RETURN %READ_EVENT_FLAG(stop_axe_flag);
ENDROUTINE setup_before_axe_case;

ROUTINE notify_after_axe_case<>;
  %SET_EVENT_FLAG(model_ready_flag);
  RETURN %READ_EVENT_FLAG(stop_axe_flag);
ENDROUTINE notify_after_axe_case;

ROUTINE axe_interface;
++
! FUNCTIONAL DESCRIPTION:
! This routine is called from the command language to
! interface to AXE and run the model
!-

! This loop executes one AXE sub-case every time through
REPEAT
  BEGIN
    ! call rtn to do setup; then check return value
    IF setup_before_axe_case() NEQ 0 THEN
      BEGIN
        %PRINT('Simulation stopped, as requested by AXE');
        %PRINT('Return from setup_before_axe_case:
            simulation stopped');
        RETURN ! to command level
      END;
    ! AXE has now initialized global section containing
    ! gpr, priviledged registers and VAX main memory
    copy_from_global_section(); ! Copy from global section
      ! to model's regs
    execute_model(); ! Run model until it hits
      ! a HALT instruction
    copy_to_global_section(); ! Copy new register values
      ! back to shared image
    IF notify_after_axe_case() THEN RETURN;
      ! Tell AXE we are done and
      ! Stop if AXE tells us to
  END;
ENDROUTINE axe_interface;

ENDBEHAVIOR;
ENDMODULE axe;

5–190
The following series of commands illustrate how to drive a DECSIM model with AXE:

$ SPANN/NOWAIT axe.com

$ DECSIM

SIM> USE chipr

SIM> SHARE/AXE vauxpr[0:15]<31:0>, vauxp[31:0], vaxmm[0:FFFFF][16]<31:0>

Network now sharable

SIM> CALL chipr.axe_interface

The following are examples of the /MAILBOX qualifier:

SIM> SHARE/MAILBOX=COMMAND
Command mailbox created: SHR164100_MBX

SIM> SHARE/MAILBOX=(COMMAND,PROMPT)
Command mailbox created: SHR164100_MBX
Prompt mailbox created: SHR164100_MBX_PROMPT
SIMULATE

Invokes network simulation execution.

FORMAT

SIMULATE [/qualifier] [ ] [integer] [units]

Default Qualifiers
/MEMORY_ERROR

Optional Qualifiers
/NOMEMORY_ERROR
/QUIET
/REPLAY
/RUNNAME=filename
/UNTIL=(stop_condition,...)

restrictions

• The SIMULATE command cannot be executed from within a watch_action.

QUALIFIERS

/MEMORY_ERROR
/NOMEMORY_ERROR

/MEMORY_ERROR is a ZYCAD–specific qualifier that causes the .ZOF file to include a description and approximate time of occurrence for each memory error. /NOMEMORY_ERROR causes the .ZOF file to show a single number totaling how many memory errors occurred.

If neither qualifier is specified, /MEMORY_ERROR is the default.

Memory errors occur when either of the following conditions is true:

• Any memory in the network is strobed for a read or write while any of the address lines are at the U state.

• Any two–state memory in the network is strobed for a write when any data input lines are at the U state.

Because designers often strobe memories more often than necessary, /MEMORY_ERROR is most useful on the initial ZYCAD run. After that, you can simply compare the number of errors with the number in the initial run.

/QUIET
Shuts off the informational message that displays the current simulation clock time.

/REPLAY
A ZYCAD–specific qualifier that produces a binary replay file with the extension .ZBR, which, when used with a restored network, allows you to replay your hard ZYCAD simulation using soft DECSIM. The filename produced by /REPLAY is the same as the network filename or the filename
specified with SIMULATE/RUNNAME. /REPLAY is valid for design verification, but is not valid for fault simulation.

SIMULATE/REPLAY is used with other commands in the following sequence:

1. SET ZYCAD
2. Either USE, COMPILE, or RESTORE
3. Any PATTERN, TRACE, WATCH, or LOAD commands
4. SIMULATE/REPLAY time
5. SAVE

You don't have to use SAVE. Instead, you can wait for the ZYCAD simulation to finish, then use SET ZYCAD/REPLAY. If you are running the job in batch, the following command:

`'VMS SYNCHRONIZE/QUEUE=ZYCAD$QUEUE runname'`

stalls the batch job until the .ZBR file is produced, at which time you can use SET ZYCAD/REPLAY without saving the network.

However, you can replay a simulation only once without a SAVE (.NSV) file. If you want to replay the simulation more than once, you must use SAVE.

Normally, WATCH and TRACE output of ZYCAD simulation is displayed as TRACE/VECTOR in the .ZOF file. However, when you use WATCH and TRACE commands prior to using SIMULATE/REPLAY, enough information is captured during the ZYCAD simulation, so that when you replay the ZYCAD simulation in soft DECSIM, the WATCH and TRACE commands have the normal full functionality.

See SET ZYCAD/REPLAY (in the section on the ZYCAD command) for information on how to use the .ZBR file produced by SIMULATE/REPLAY.

/.RUNNAME

A ZYCAD--specific qualifier that allows you to specify the file name used in naming the .ZIF, .ZTF, .ZOF, .STT, and .OUT files. If you don't use the /RUNNAME qualifier, the default file name is the network file name.

The file name can be specified in any of the following ways:

- `SIMULATE/RUNNAME=name`
- `SIMULATE/RUNNAME={area}name`
- `SIMULATE/RUNNAME=disk$:{area}name`
- `SIMULATE/RUNNAME=logical:file`

The file name must be specified with no file extension because it is used in naming several types of files.

If disk and area information are specified, the files are written to the specified area. The default area is the current directory at the time of simulation.

The run name (file name) is printed out in the .ZOF file on the same line as, and immediately following, the run type (Design Verification or Fault Run).
\textbf{UNTIL = stop\_condition}

Simulates until the stop\_condition is satisfied. Stop\_condition can be:

- \texttt{[NO]GOOD\_IDLE} – simulation continues until there are no more good events to simulate. \texttt{GOOD\_IDLE} is the default.
- \texttt{[NO]PATTERN\_IDLE} – simulation continues until all pattern files have been closed. Pattern files are closed when the last pattern has been read but not executed.
- \texttt{ALL} – sets the above conditions as stop\_conditions.
- \texttt{NONE} – specifies that there is no stop\_condition. Simulation continues until you explicitly stop it with a \texttt{WATCH}, \texttt{CTRL/C}, \texttt{BREAK}, etc.

More than one stop\_condition may be specified by separating the conditions with commas and enclosing the conditions in parentheses (for example, \texttt{SIMULATE/UNTIL=(GOOD\_IDLE, PATTERN\_IDLE)}. If more than one condition is specified, simulation continues until all of the conditions have been satisfied. If a time is also specified, simulation continues until the specified time or until the other stop\_condition is satisfied. For example, \texttt{SIMULATE/UNTIL=GOOD\_IDLE 1000} simulates until there are no more good events to simulate or for 1000 time units.

\section*{DESCRIPTION}

The \texttt{SIMULATE} command begins actual simulation evaluation of network components and propagation of network values and states.

As shown above, there are various conditions that can be specified on the command line to terminate simulation. Simulation can also be stopped with \texttt{CTRL/C}, which prints the \texttt{SIMULATE} command being executed and the current simulation time. Option "X" on the menu that is printed after typing \texttt{HELP} at the \texttt{CTRL/C} prompt prints this \texttt{SIMULATE} command again.

\texttt{SIMULATE} with no arguments continues simulation execution until no more events are pending or until the network is idle. Whether or not the network is idle is determined by gate activity. External events pending due to a \texttt{DEPOSIT} or \texttt{PATTERN} command are not considered in evaluating if the network is idle. If a previous \texttt{SIMULATE} command specified a time that has not yet passed for this command, the simulation halts at that time.

\texttt{SIMULATE} begins with the current time slot before it advances time. Thus, when \texttt{SIMULATE 0} is typed, zero-delay activity (and \texttt{DEPOSIT}s) are executed for the current time slot only.

Signal activity conditions can be simulation termination conditions if the conditions are entered as a \texttt{WATCH} or \texttt{BREAK} (a command macro) command. Thus both a simulate signal condition and time-out can be obtained.

The time specification may be given as relative or absolute. An absolute time is preceded by a vertical bar (\texttt{|}).
**ZYCAD Mode**

In ZYCAD mode, the SIMULATE command submits the job to the ZYCAD unit.

1. DECSIM writes a ZYCAD input file (ZIF) from the information provided in the session.
2. When the ZIF is written to disk, the SIM> prompt returns.
3. The ZIF is copied from your system to the system with the ZYCAD unit.
4. The system with the ZYCAD unit does the simulation, creates the ZYCAD output file (.ZOF), and ships back the output.
5. You receive notification, usually a mail message, when your job is completed.

For more information see:

<table>
<thead>
<tr>
<th>BREAK</th>
<th>HALT</th>
<th>PAUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTINUE</td>
<td>INDIRECT</td>
<td>TIME</td>
</tr>
<tr>
<td>CTRL/C</td>
<td>LOG</td>
<td>ZYCAD</td>
</tr>
</tbody>
</table>

**EXAMPLES**

SIM> SIM
SIM> SIMUL 0
SIM> SIMULATE 1000 NS
SIM> SIMULATE/UNTIL~(PATTERN_IDLE, GOOD_IDLE) 2000
SPAWN

Invokes VAX/VMS as a subprocess. When used alone, VAX/VMS displays the dollar sign prompt ($), indicating that it is ready to accept any DCL command as input. When used with a command, the command is executed and then control is returned to DECSIM.

FORMAT

SPAWN [/ qualifier] [DCL_command_string]

where DCL_command_string is any valid DCL command.

Default Qualifiers

/KEYPAD
/LOGICAL_NAMES
/NONOTIFY
/SYMBOLS
/WAIT

Optional Qualifiers

/INPUT=filename
/NOKEYPAD
/NOLOGICAL_NAMES
/NOSYMBOLS
/NOTIFY
/NOWAIT
/OUTPUT=filename
/PROCESS=subprocess_name

restrictions

None.

ARGUMENTS

DCL_command_string

The DCL_command_string can be any valid DCL command.

QUALIFIERS

/INPUT = filename

Specifies an input file containing one or more DCL commands to be executed by the spawned subprocess.

/KEYPAD
/NOKEYPAD

Specifies whether keypad symbols and the current keypad state are copied from the parent process to the subprocess. /KEYPAD is the default.

/LOGICAL_NAMES
/NOLOGICAL_NAMES
Determines whether the system passes process logical names and logical name tables to the subprocess. /LOGICAL_NAMES is the default. (See the SPAWN command in the DCL Dictionary for exceptions.)

/NOTIFY
/NONOTIFY
Controls whether a message is sent to the terminal notifying you that your subprocess has been completed or aborted. /NOTIFY is often used with /NOWAIT, but it may not be used when the SPAWN command is executed from within a noninteractive process. /NONOTIFY is the default.

/OUTPUT = filename
Specifies that the output from the SPAWN operation will be written to the specified file.

/PROCESS = subprocess_name
Specifies the name of the subprocess to be created.

/SYMBOLS
/NOSYMBOLS
Determines whether DCL global and local symbols are passed to the subprocess. /SYMBOLS is the default.

/WAIT
/NOWAIT
Controls whether the system waits until the current subprocess is completed before allowing more commands to be issued in the parent process. /NOWAIT allows you to issue new commands while the specified subprocess is running. /WAIT is the default.

**DESCRIPTION**
The SPAWN command allows you to run a subprocess without exiting DECSIM.

Typing SPAWN by itself will give you the DCL dollar sign prompt ($). Any valid DCL command may then be entered. Type LOGOUT to return to the SIM> prompt.

A DCL command may be entered after SPAWN. After the DCL command is executed, control is returned to DECSIM. The /INPUT qualifier may be used to specify an input file containing several DCL commands to be executed by the subprocess.

The SPAWN command works in ZYCAD mode.
EXAMPLES

SIM> SPAWN
%I-VCONT, Type LOGOUT to return to DECSIM.
$

SIM> SPAWN EDIT test.log
Input file does not exist
[EOB]
$

SIM> SPAWN/PROCESS = mike/NOTIFY/OUTPUT = test.log SHOW TIME
Subprocess MIKE has completed.
SIM> SPAWN TYPE test.log
  9-SEP-1987  08:46:15

SIM>
**SPR (Command Macro)**

The SPR command reports problems and questions to the DECSIM developers.

**FORMAT**

<table>
<thead>
<tr>
<th>SPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Qualifiers</td>
</tr>
<tr>
<td>None.</td>
</tr>
<tr>
<td>Optional Qualifiers</td>
</tr>
<tr>
<td>None.</td>
</tr>
</tbody>
</table>

**DESCRIPTION**

The SPR command invokes the DECSIM software problem report facility in DECSIM. Your SPR contains your user name, the date and time, and the version of DECSIM you are running. DECSIM reserves the top few lines of your screen as a no-scroll section to display questions for you to answer. You answer the questions as directed and then complete the SPR with CTRL/Z.

CTRL/Z sends the SPR to the system developers via the VMS NMAIL facility and returns you to the SIM> prompt, ready to continue with your simulation. You also receive a copy of your mailed SPR.

If, for some reason, the VMS MAIL facility is unable to send your SPR to the DECSIM Release/Support team, then either try to forward the SPR (using the VMS MAIL facility) at a later time, or send a hard copy to the DECSIM Release/Support team. Keep your copy of the SPR until you receive verification from DECSIM or the DECSIM Release/Support team that the SPR has been received and logged in. If you need to check on an SPR, contact SEG/CAD DECSIM Release/Support team (HL2-2/H13 DTN 225-5612).

For more information see: CONFIGURE

**SPR Macro Definition**

```plaintext
SET MACRO /COMMAND /ABBREV=2 SPR =
log/logname=log decsimspr.tmp
cancel log log
vms @sys$decsim:spr.com
$ENDMACRO
```
STATE

Creates variables in the command language context.

**FORMAT**

[SET] STATE name [,name... ]
CANCEL STATE name [,name... ]
SHOW STATE

**Default Qualifiers**

None.

**Optional Qualifiers**

None.

**restrictrons**

- The variable can be no larger than 256 bits.

**ARGUMENTS**

name

A new signal name with optional word and bit subscripts, but no hierarchy. The name should not be the same as any name appears in the network description. Only names that have been declared in a STATE command (as opposed to network names) may be canceled. Names may not be abbreviated. STATE names may include the underscore (_) character.

**DESCRIPTION**

SET STATE defines command language variables that are in addition to the signals defined in the network. The STATE command is not to be confused with behavior language state variable declarations. You can think of command language STATE variables as integer variables in most programming languages. They can have both bit and word subscripts for bit vectors and word arrays. If a name defined by the STATE command corresponds to a real signal name or behavior STATE variable name in the network description, the command language STATE variable will be used in all expression and EVALUATE accesses. Exceptions to this are DEPOSIT targets and EXAMINE arguments. You cannot declare the prefix %NET on a command state. Command language STATE variables are initialized to 0.

STATE names may be used in any DECSIM expression. There is no scoping (for block levels) for command language defined STATE variables; all are considered global for command access and are always accessible no matter what the network SCOPE setting is. STATES are not affected by the SCOPE setting.

CANCEL STATE deletes a STATE name; word and bit subscripts are ignored in CANCEL STATE.
A name can be assigned a value in an EVALUATE command. If the name has not been previously declared in a STATE command, EVALUATE will implicitly define it and give a warning message. This does not happen if the undefined target name had subscripts. Subscripts are not checked for overlapping values.

For more information see: EVALUATE

EXAMPLES

SIM> STATE A, B[0:5], C<15:0>, D[1:10]<23:0>
SIM> SHOW STATE
A<31:0> = 0
B[0:5]<31:0>
C<15:0> = 0
D[1:10]<23:0>
STATISTICS

Prints certain statistics on DECSIM runtime, network size, etc.

FORMAT

[SHOW] STATISTICS [/QUALIFIER...]

Default Qualifiers
None.

Optional Qualifiers
/LOGNAME=logname
/LOGNAME=(logname,...)
/NETWORK
/SIMULATION

restrictions
None.

QUALIFIERS

/LOGNAME = logname
/LOGNAME=(logname,...)
Routes the STATISTICS information to a log file. If a logname has been previously set up with a LOG command, output is routed to this log file. Otherwise, an implicit SET LOG command is performed, which opens a log file. The log file remains open until you explicitly close it with CANCEL LOG. Multiple lognames can be used to print to multiple files by enclosing the list in parentheses.

The logname specified is a logical name, and the resulting file has the extension .LOG.

/NETWORK
Prints statistics pertaining to the currently loaded network.

/SIMULATION
Prints a table of information, in landscape mode (132 characters wide), in the following order:

• Primitive elements
  Total number
  Scalar elements
  Register transfer level elements (phantom splits, merges, and read and write memory primitives)
  MOS charge sharing capacitors
  Read memories
  Write memories

• Fault inserted
  Total number
  Scalar faults
• Events pending
  Total number
  Scalar events
  Register transfer level events
  MOS charge sharing events (node changed to 1#2 due to charge sharing)
  MOS time out events (node changed to U#2 due to time elapsed)
  MOS regular event (node driven to 1#2 by a transistor)
  Read memory event
  Write memory event
  Behavior event

• Events canceled – same categories as events pending

• Events executed – same categories as events pending

• Events executed since the last STATISTICS command that was entered, shown as "Evt log" – same categories as events pending

• Fault events pending – same categories as events pending

• Fault events canceled – same categories as events pending

• Fault events executed – same categories as events pending

• Fault events executed since the last STATISTICS command that was entered, shown as "F Evt exe" – same categories as events pending

• Faults diverged (fault effects created)
  Total number of faults
  Scalar faults
  Register transfer level faults
  MOS charge sharing faults
  Write memory faults

• Faults dropped – same categories as faults diverged

• Faults present – same categories as faults diverged

At the base of the table, information is given on the number of primary inputs, the number of split and merge elements, that is, the number of phantoms inserted, the number of page faults, and the number of pages used in memory. It also gives the events per CPU second, the fault events per second, the percentage of faults detected, the number of undefined states, the CPU time elapsed, and the simulation time. In addition, it shows the number of detection points, the number of firm detections, the number of possible detections, the number of inserted faults canceled by /LIMIT, and the average faulty machine size.

**DESCRIPTION**

Provides statistics for the currently loaded network, such as the number of defined behavior states, the number of gates, the sum of gate inputs and outputs, etc.
SIM> USE mixed_fulladdert
    Reading File: DOC$DISK:[USER.PROJECT]MIXED_FULLADDETR.NOB;1
    Reading File: DOC$DISK:[MCGEE.REFMAN]MIXED_FULLADDETR.EXE;1
%I-TRS, Time Resolution = 1 nS, Default unit is nS
SIM> SHOW STATISTICS
--- GENERAL statistics ---
  Simulation Time = 10 nS, Time Resolution = 1 nS, default unit is nS
  Last SIMULATE (CALL) command ran 0.020 Cpu-Seconds,
  PRIMARY Network idle
  10 "U" outputs on network elements
SIM> SHOW STATISTICS/NODE
  NETPRO Library Format Rev: 22 for VAX
  Network Name: NETWORK, Rev: 0
  Source File: DOC$DISK:[MCGEE.REFMAN]MIXED_FULLADDETR.NOB;1

  Zoom Primitive:Rev     Inputs
  NOT:0                  1
  AND2:0                 2
  OR2:0                  2

  Behavior MODELS         1
  Behavior STATES         1
    _Compiler-Gend Temp   0
  EQUATION Primitives    5
    _Phantom EQN Primitives 4
  Node Count             18
    _Phantom Node Count   8
  MOS Nodes               0
    _Bidirectional MOSFETS 0
    _Unidirectional MOSFETS 0
    _Stacks                0
  Structural MEMORIES    0
    _Memory READ Ports    0
    _Memory WRITE Ports   0
    _Memory READ/WRITE Ports 0
  Split/Merge Elements   4
  VECTOR Outputs         0
  VECTOR Inputs          0
  SCALAR Primary-Inputs  3
    (Unconnected)        0
    (Single Fanouts)     0
STEP

Terminates the current interactive debug session, returns to simulation at the point at which it was interrupted, and steps through (executes) a specified number of behavior language lines. Then the debug session is reentered. STEP is functionally equivalent to the CANCEL BREAK /STEP command.

FORMAT

STEP [step_count]
STEP /INSTRUCTION [step_count]

Default Qualifiers
None.

Optional Qualifiers
/INSTRUCTION

restrictions

- The STEP command can be used only when you enter debug mode with the WATCH/STATEMENT command. It is not possible to step forward from any other type of WATCH trigger, including WATCH/CALL.

ARGUMENTS

step_count
Specifies the number of behavior language source lines to execute before returning to interactive debug mode. The default is 1.

The step_count pertains only to the statements in the currently executing routine. A call to another routine counts as one statement, regardless of how many statements may be executed in the called routine. Stepping outside the routine is not allowed. Debug mode will be reentered automatically when the end of the routine is reached, even if the step_count has not yet been exhausted.

QUALIFIERS

/INSTRUCTION
Specifies that the single-stepping action units are VAX machine instructions rather than behavior language source lines.

For more information see:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANCEL BREAK</td>
<td>ENABLE BREAK</td>
</tr>
<tr>
<td>DISABLE BREAK</td>
<td>SET BREAK</td>
</tr>
<tr>
<td></td>
<td>SHOW</td>
</tr>
<tr>
<td></td>
<td>RESUME</td>
</tr>
</tbody>
</table>
EXAMPLES

SIM> WATCH /STATEMENT=9 DO BREAK
%I, [SIMTXT] Watchpoint set at /STATEMENT:E_MOD\9 (000DCE1B)
%I, [SIMWNT] WATCH/TRACE Name is WATCH1
SIM> CALL rtna
BREAK from WATCH /WATCHNAME:WATCH1 /STATEMENT:E_MOD\9 (000DCE1B)
SIMBREAK> STEP
BREAK after stepping to line 10
SIMBREAK> STEP 999
BREAK after stepping to end of routine at /STATEMENT:E_MOD\18 (000DCE9C)
SIMBREAK> RESUME
SIM>
SYMBOLS (Command Macro)

Allows you to look at the DECSIM symbol table and search for signal names and values. Names are printed in alphabetical order.

**FORMAT**

<table>
<thead>
<tr>
<th>SYMBOLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Qualifiers</td>
</tr>
<tr>
<td>None.</td>
</tr>
<tr>
<td>Optional Qualifiers</td>
</tr>
<tr>
<td>None.</td>
</tr>
</tbody>
</table>

**restrictions**

None.

**DESCRIPTION**

The SYMBOLS command prints an alphabetically ordered list of all the signal names and current values. The range of the current setting of SCOPE is used.

For more information see:

EXAMINE  SCOPE  SHOW WATCH

**EXAMPLE**

```
SIM> SYMBOLS
A = 1
CLK = 0
DATA = U
SET = 0
SIM>
```

**Macro Definition**

```
SET MACRO /COMMAND /ABBREVIATION=3 SYMBOLS = EXAMINE "*" $ENDMACRO
```
Tells DECSIM what kind of terminal you are using and allows you to change the terminal type.

**FORMAT**

**[SET] TERMINAL**  
**[terminal_type]**

**SHOW TERMINAL**

**ARGUMENTS**

**terminal_type**  
Terminal_type is one of the following: VT05, VT50, VT52, VT100, LA36, LA120, TTY, BATCH.

**DESCRIPTION**

If no argument is given, SET TERMINAL queries the operating system for the type. This results in an implicit automatic setting of the terminal characteristics.

**Note:** The SET TERMINAL command sets the VT200 series of terminals to VT100.

Foreign terminals and line printer files should use "SET TERMINAL TTY".

You can include the TERMINAL command in the DECSIM.INI or netname.INI initialization file, but SET TERMINAL is ignored in a batch job. The DECSIM command language adapts itself in a few respects to the type of terminal. The following types are supported:

**hard copy:**

- For batch, LA36, LA120, etc.
- (No special graphics characters)

**display:**

- For VT05, VT52, VT100, VT200, etc.
- (Some escape sequences for cursor control)

Special qualifiers of the EXAMINE, TRACE, and DEBUG commands utilize the video screen features when terminal characteristics are set to allow it.

Older style terminals (such as LA30's and VT05's) can be used, but some special characters, such as [ and ], are not available. Command mode makes little or no use of these characters, however.
In certain tracing modes, the upper part of the screen can be reserved for special "odometer" style signal value reports, which leaves the lower portion of the screen for normal use. Some features are software-simulated on a VT52 where a VT100-specific capability is used.

For more information see:

- EXAMINE
- PRINT
- Initialization
- TRACE

**EXAMPLES**

```
SIM> SET TERMINAL VT52
SIM> SHOW TERMINAL
Terminal type: LA120
```
TIME

Displays the current value of the simulator clock along with the current resolution units, and resets the simulation time to a specified absolute time.

FORMAT

[SHOW] TIME
SET TIME | time [units]
SET TIME/UNITS=units !(for input and output)
SET TIME/[NO]ODOMETER

Default Qualifiers
/NOODOMETER
/UNITS=NS

Optional Qualifiers
/ODOMETER
/UNITS=units

restrictions
None.

ARGUMENTS

units
Units is one of: FS, PS, NS, US, MS, S.

DESCRIPTION

The SHOW TIME command displays the current simulator clock time with the current time units (nanoseconds, picoseconds, etc). Time is always shown in radix 10, no matter what default is specified with the SET RADIIX command.

The SET TIME command is used for three purposes:

1. SET TIME (without any qualifiers) resets the time to the specified absolute time (usually 0). This number or offset is used when interpreting times on inputs and outputs. For this case, the network must be completely idle. If the network is not idle:
   - All scheduled events will mature at the same “time” but will be reported using the new offset.
   - Commands handle unscheduled events inconsistently.

The time is considered to be absolute whether or not 1 precedes the time specification. Spaces between time and time units are optional.

Note: SET TIME 0 resets only the time reference. It does not reset the state of the simulator to what it was at time 0.
2 The SET TIME/UNITS=units command chooses the printout units for all DECSIM time reports. It also changes the default units for input. This is seen in the PRINT %TIME command, SHOW TIME, and simulation time reports.

3 SET TIME /ODOMETER displays the simulator clock time on a video terminal during simulation. It is not updated during command mode—only during simulation execution.
TIMING

Specifies which of the three timing parameters is loaded by the USE and COMPILE commands: min, nom, or max.

FORMAT

[SET] TIMING
SHOW TIMING

Default Qualifiers
/MAX

Optional Qualifiers
/MIN
/NOMINAL

restrictions
None.

ARGUMENTS
None.

QUALIFIERS

/MIN
/NOMINAL
/MAX

Specifies the parameter triple that DECSIM uses when you load your network. The default is /MAX.

DESCRIPTION

The TIMING command allows you to specify which of the three parameter triple values to use when loading the network. The TIMING command must be used before the USE or COMPILE command; it does not affect loaded networks.

TIMING can be applied to equation primitives, memory delays, and, on a limited basis, to behavior models. See Section 3.5 for more information.
EXAMPLE

SIM> TIMING /MIN
%H-TAU, TIMING will affect next USE (COMPILE) command
SIM> SHOW TIMING
    MIN    /DEFER:MAX

The phrase "/DEFER:MAX" refers to an unimplemented feature

SIM> TIMING/MAX
%H-TAU, TIMING will affect next USE (COMPILE) command
SIM> SHOW TIMING
    MAX    /DEFER:MAX
SIM> TIMING/NOMINAL
%H-TAU, TIMING will affect next USE (COMPILE) command
SIM> SHOW TIMING
    NOMINAL    /DEFER:MAX
Controls specialized reporting of various network transitions. You can route signal activity traces to an ASCII- or binary-formatted log file, or to the terminal.

TRACE can be used with vector items such as behavior and memory ports. It can also be used with behavior states and arrays, both global and local. It can monitor a subfield of a subscripted port or state. For example, given a port A<7:0>, it can trace A<2>, and given an array A[0:3]<7:0>, it can trace a single element A[1]. It cannot monitor a memory primitive array or a command language state.

TRACE has four formats and many of the same qualifiers as the SET WATCH command. Refer to Table 5-9 to see which qualifiers apply to the four formats.

The TRACE variants prefixed with CANCEL/DISABLE/ENABLE and SHOW are described as separate commands following the SET TRACE command description.

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>[SET] TRACE [ /qualifier... ] access_list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Qualifiers</td>
<td></td>
</tr>
<tr>
<td>/ACTIVITY</td>
<td></td>
</tr>
<tr>
<td>/FORCED</td>
<td></td>
</tr>
<tr>
<td>/FROM=0</td>
<td></td>
</tr>
<tr>
<td>/IMMEDIATE</td>
<td></td>
</tr>
<tr>
<td>/LOG=logname.TRA</td>
<td></td>
</tr>
<tr>
<td>/NOCOMPRESSED</td>
<td></td>
</tr>
<tr>
<td>/NORMAL</td>
<td></td>
</tr>
<tr>
<td>/TO=INFINITY</td>
<td></td>
</tr>
</tbody>
</table>

| Optional Trace Execution Qualifiers |
| /DELAYED |
| /EVERY=time |
| /EVERY=[time_list] |
| /EVERY=(time_expression) |
| /EXECUTE |
| /FROM=[ ]time[units] |
| /RETURN=routine_name |
| /TO=[ ]time[units] |
| /UNTIL=(expression) |
| /WHEN=(expression) |
| /WHILE=(expression) |

| Optional Trace Format Qualifiers |
| /BINARY |
| /CALL=routine_name |
| /COMPRESSED |
| /EXACT |
| /FULL |
/GRAPH
/HEADER
/LOGNAME=logname
/LOGNAME=(logname,...)
/MAP=(original_name=resulting_name)
/ODOMETER
/RADIX=integer
/REPLAYABLE
/SEPARATOR='text'
/TRACENAME=trace_name
/UNFORCED
/VECTOR
/WIDTH=integer
/ZEROFILL

restrictions

• You cannot TRACE behavioral states in conjunction with the SHARE command due to the conflict of protecting the global section.

• Further restrictions apply in ZYCAD. See DESCRIPTION below.

QUALIFIERS

/ACTIVITY
Prints the signal value every time there is a transition. /ACTIVITY is the default for /NORMAL and /ODOMETER in all cases and the default for /VECTOR and /GRAPH unless some other trigger is specified.

/BINARY
Creates a compact binary log file of the trace data. There are two forms of the file: /VECTOR and /NORMAL. /VECTOR is the default.

/VECTOR and /NORMAL have the same functionality with /BINARY as they do without /BINARY. /VECTOR records the state of all the nodes specified in the TRACE command; /NORMAL records only the state of the nodes that change states. See the /VECTOR and /NORMAL qualifiers for more details. Both forms support the use of double-precision time values.

You must use /LOGNAME with /BINARY. It is not necessary to open a log file with the SET LOG command; the /LOGNAME qualifier opens a log file if a file of that name does not exist. (See /LOGNAME for details on how to open a trace log file.)

The most common uses of binary trace files are to create a replayable file to use with the REPLAY command, to create a binary formatted file to use as a pattern file with the PATTERN/BINARY command, or to create an input file for a postprocessor. See TRACE/REPLAYABLE and PATTERN/BINARY for more information. If you intend to use the binary file as a pattern file in a network that uses different signal names, you must use the /MAP qualifier to alter the names. See /MAP below for more details.

If you want to use the binary file for other purposes, see Appendix G, which describes in detail how to create, read, and extract information from binary trace files.
/CALL=routine_name
Triggers the trace when the specified behavior routine begins execution. The trace is triggered after the actual parameters are copied but before the local states are initialized.

/COMPRESSED
/NOCOMPRESSED
/COMPRESSED is a ZYCAD-specific qualifier that compresses the format of the .ZOF file so that there are no spaces between the traced signals, and there is only one header for each horizontal page. The .ZOF file displays the results of the specified signals for specified simulation times. Those signals that do not fit on the page are reported, just like the other signals, on a new page (horizontal page). To read the .ZOF file, place the "horizontal page" for each succeeding set of signals to the right of the previous one, to form a page wider than a printer can handle.
/NOCOMPRESSED is the default qualifier. DEC3IM accepts the format specified by the last TRACE command entered before the job is submitted to the Logic Evaluator as the report format.

/Delayed
Specifies that action commands are performed at the end of the specified time slot. This is the default for /GRAPH and /VECTOR. When applied to local behavior states, /DELAYED is interpreted as /IMMEDIATE, and no warning is given.

/EVERY=time
/EVERY=[time_list]
/EVERY=(time_expression)
Indicates that the trace is triggered by the simulator clock. Times can be absolute or relative. If a time list contains one or more absolute times, all times in the list are considered to be absolute. Otherwise, times are relative. A time list consists of times separated by commas. The list is enclosed in square brackets. A time expression can also be used. Enclose the expression in parentheses (for example, /EVERY=(5+A)). Times must be given in chronological order with no 0 values for relative times or repeated absolute times. Triggering is repeated for a relative time list. /IMMEDIATE is not applicable with this qualifier.

/EXACT
Prints the form of the signal name as you gave it in the access list, rather than translating it to the full hierarchical name. This is the default for /NORMAL and /ODOMETER traces. When the /HEADER qualifier is used with /VECTOR and /GRAPH, signal names are also printed as you gave them in the access list, unless you use the /FULL qualifier.

/EXECUTE
Triggers a trace immediately, that is, in the first time slot of simulation. The command TRACE/GRAPH/EXECUTE signalname or TRACE/VECTOR/EXECUTE signalname forces the immediate printing of current values in these types of traces; it is ignored in other types of traces. If you have specified a header for the TRACE command, it is printed with the values.
/FORCED

/UNFORCED
/UNFORCED allows you to monitor the value a node would have had if it had not been forced by the DEPOSIT/FORCE or PATTERN/FORCE command. This does not work for MOS nodes forced to a high impedance (Z#2) state. If there is no forced value for the node, the actual value (value of the driving output) is displayed, no matter which of the two qualifiers you use.

/FROM=[ ]time[units]
Specifies a starting time for the TRACE command.

When used in ZYCAD mode, specifies when the Logic Evaluator starts output reporting. The default, if the /FROM qualifier is not specified, is to start output reporting at time zero. This qualifier is useful for skipping over unwanted network initialization data.

/FULL
In /NORMAL and /ODOMETER traces, this qualifier prints the full hierarchical name of the signal name, from the currently defined SCOPE down to the primitive level. For /FULL to work in /GRAPH and /VECTOR traces, you must also use the /HEADER qualifier.

/GRAPH
Produces a representation in graph form of the traced signal values. An undefined (U) or a high impedance (Z) is shown literally. The “1” or “0” states are depicted positionally. Each graph trace line begins with the time the values are being reported. /ACTIVITY is the default. Can be used only on a single-bit vector or on a single-bit subfield of a larger vector. For behavior models, this qualifier can trace only a single bit of a single element of an array, or a single bit of a state. For example, given an array A[0:3]<0:7>, you can trace A[1]<3>, but you cannot trace A.

/HEADER
The /HEADER qualifier prints a heading identifying the signal names specified for /GRAPH and /VECTOR. This qualifier is ignored for /NORMAL and /ODOMETER.

/IMMEDIATE
Executes the TRACE action right after the transition has matured (this is the default for /NORMAL and /ODOMETER). When /IMMEDIATE is used with /GRAPH or /VECTOR, it causes the printing of one line for each transition instead of a single summary line containing the state at the end of the time slot.

/LOGNAME= logname

/LOGNAME= (logname,...)
Specifies destination of trace data. If a log file has previously been opened by a LOG/TRACE command, TRACE output is routed to this logname. Otherwise, this qualifier executes an implicit SET LOG command and opens file logname.TRA. If the /BINARY qualifier is also specified, the file format is binary, not ASCII.
To append information from a trace to an existing binary trace file, execute a SET LOG/TRACE/NOHEADER/APPEND command first, then reference the log file with the /LOGNAME qualifier on the TRACE command.

/MAP=(original_name=resulting_name)
Specifies new names (resulting names) to be recorded in the binary format trace file in place of the original signal names in the access list.

The /MAP qualifier can be used with the /BINARY, /VECTOR and /LOG qualifiers when you intend to use the binary trace file as a binary pattern file to drive the inputs of another network whose signal names do not match the signal names in your current network. See the PATTERN/BINARY command for more information on driving a network with a binary file.

The syntax and rules that apply to signal names apply to the original names and resulting names. A resulting name can also be %NULL.

The wildcard characters (*) and (?) can be used in original names and resulting names. The rules for mapping original names with resulting names when wildcard characters are used are as follows:

- A resulting_name is matched left-to-right with the wildcard characters in an original name.
- Mappings are attempted in the order specified in the argument to /MAP qualifier, and the first successful match is the one used.
- If no match is found, the original name is used.

For example:

TRACE/BINARY/VECTOR/LOGNAME=bipat.tra/MAP=(*%p=*) in1%p, in2%p, tmp1

In this example the signal names in the access list whose names end with the suffix "%p" have this suffix removed when they are recorded in the binary file.

The /MAP qualifier can be used to transform vectors into vector or scalar subsets. In these transformations the following rules apply:

- Signals that are not specified as original names in a /MAP qualifier argument have no alterations to their subscripts.
- The entire subscript bit width for an original name is copied to its resulting name unless a subscript is included with an original name to specify a bit or bit range.
- The wildcard character (*) can be used in subscripts for original names. The asterisk (*) substitutes for a single bit number. For example:

  /MAP:(%c<>=_.*)
  Removes the "%c" suffix, converts the bit numbers to a suffix that begins with an underscore and the a bit number.
  A signal named xyz%c<7:0> is transformed to a list of signals that are named xyz_7, xyz_6, xyz_5, xyz_4, xyz_3, xyz_2, xyz_1, and xyz_0.
/MAP:(+%<*:5>=ْ*) Transforms three of the eight bits in a signal named xyz%<7:0> to signals named xyz_7, xyz_6, and xyz_5.

/MAP:(+%<3:=ْ*) Transforms four of the eight bits in a signal named xyz%<31:0> to signals named xyz_03, xyz_02, xyz_01, and xyz_00.

- A subscript for an original name is not automatically copied to the resulting name. If no subscript is specified for a resulting name it will have no assigned bit number or numbers but it will have the same bit width as the original name. You can specify a subscript for the resulting name; if you do, the bit width of the resulting name must match the bit width of the original name. For example:

/MAP:(<0:=ْ>) Removes the <0> from the name.

/MAP:(<5:=ْ5>) Converts the <5> subscript to a "_5" suffix.

/MAP:(<31:16:=ْ<15:0>) Changes the bit numbers but not the bit width.

- If you specify a subscript for a resulting name without including a subscript for the original name, the subscript must be for a bit width of 1.

/NORMAL
Prints a single signal, variable, or memory location when it changes.
Includes time, old value, and new value. This qualifier forces /IMMEDIATE mode, which means the report is done when the transition occurs and not at the end of the time slot. The /DELAYED qualifier is ignored if used in combination with /NORMAL.

/ODOMETER
Reserves lines at the top of the screen as nonscrolling area. The current value and time of last transition are displayed in reverse video. For behavior models, this qualifier traces only a single element of an array. For example, given an array A[0:3]<7:0>, you can trace A[1], but you cannot trace A.

/RADIX=integer
Defines the radix used to display port and state vector values. Scalar values are always displayed in radix 2. If this qualifier is not given, the radix is defined by the SET RADIX command.

/REPLAYABLE
Creates a compact, binary file of the trace data that you can play back to review a simulation. TRACE/REPLAYABLE is synonymous with TRACE/BINARY/VECTOR/DELAY. You must use /LOGNAME with /REPLAYABLE to create the replayable log file.

To replay the simulation data, you use the REPLAY/FILE command. After entering the REPLAY command, you can set time backwards or forwards and examine any of the traced signals. See the REPLAY command for the procedure and an example.
/RETURN=routine_name
Triggers the trace when the specified behavior routine exits. Any return
value in the routine has been updated in a local state in the calling
routine, a global state in a model, and always in the system variable
%ROUTINEVALUE. (Note that %ROUTINEVALUE system variable
is accessible only from the command language and not from behavior
models.)

/SEPARATOR='text'
Puts 'text' separator in between each traced output value. Default
is ' ' for /GRAPH. The default separator for /VECTOR is the null
string. /SEPARATOR is ignored for /ODOMETER and /NORMAL. If
/SEPARATOR='tab' is given, DECSIM assumes the standard 8, 16, 24,...
tab setting when computing the line length.

/TO=[.]time[units]
Specifies the time to end the TRACE reporting of the signals.

/TRACENAME=trace_name
Supplies a trace label for referencing this TRACE command in future
ENABLE and DISABLE or CANCEL TRACE commands. If omitted, a
default unique name is supplied. SHOW TRACE shows the trace_names.

/UNTIL=(expression)
The expression specified by /UNTIL is tested whenever any of its
components change, and if the value is true, the trace is canceled. The
expression evaluation happens at the end of the time slot. /UNTIL is not
affected by the /IMMEDIATE qualifier.

/VECTOR
Reports the logic values of all traced signals in a single–line format. Each
vector trace line begins with the time the values are reported. The values
are printed as specified with the /SEPARATOR, /RADIX, /WIDTH, and
/ZEROFILL qualifiers. Default is /ACTIVITY. For behavior models, this
qualifier traces only a single element of an array. For example, given an
array declaration A[0:3]<7:0>, you can trace A[1], but you cannot trace A.

/WHEN=(expression)
Every time the /WHEN expression becomes true, the trace is triggered.
Even though the expression operands may change, the trace is triggered
only when the expression result changes and the new (low–order bit) value
is true. For these purposes, U and Z results are considered to be false.

/WHILE=(expression)
The expression specified by /WHILE enables tracing when the expression
is true, and disables tracing when false. The expression evaluation
happens at the end of the time slot in which one or more of the expression
operands changed. /WHILE is not affected by the /IMMEDIATE qualifier.

/WIDTH=integer
For /VECTOR traces, specifies the number of characters allocated to print
a value. The value in this field is right justified. A value requiring a wider
field overflows the width specified.
/ZEROFILL
For /VECTOR traces, specifies that the value in the field defined by
/WIDTH be right justified and filled with zeros. The default is to fill the
field with spaces. This qualifier is not applicable for /BINARY.

DESCRIPTION
SET TRACE establishes a set of signals and behavior STATE variables
to be traced. It also sets the format and style in which the report will be
made. There are four TRACE formats: /NORMAL (the default), /VECTOR,
/GRAPH, /ODOMETER.

The following table describes the output of the four TRACE formats.

<table>
<thead>
<tr>
<th>Table 5–6 TRACE Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
</tr>
<tr>
<td>/NORMAL</td>
</tr>
<tr>
<td>/VECTOR</td>
</tr>
<tr>
<td>/ODOMETER</td>
</tr>
<tr>
<td>/GRAPH</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

The following table shows which formats will trace the various types of
signals.

<table>
<thead>
<tr>
<th>Table 5–7 Trace Formats and Signal Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal Type</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>scalar</td>
</tr>
<tr>
<td>behavioral</td>
</tr>
<tr>
<td>port, state</td>
</tr>
<tr>
<td>behavioral</td>
</tr>
<tr>
<td>array</td>
</tr>
<tr>
<td>memory port</td>
</tr>
<tr>
<td>memory</td>
</tr>
<tr>
<td>primitive array</td>
</tr>
<tr>
<td>MOS node</td>
</tr>
<tr>
<td>command</td>
</tr>
<tr>
<td>language state</td>
</tr>
</tbody>
</table>

Key

yes–the format traces this type of signal
no–the format does not trace this type of signal
bit–the format traces only one bit of the signal
elem–the format traces only one element of the array
The following table shows what TRACE execution qualifiers will work in conjunction with the four formats.

**Table 5-8  TRACE Execution Qualifiers**

<table>
<thead>
<tr>
<th>TRACE</th>
<th>/NORMAL</th>
<th>/ODOMETER</th>
<th>/VECTOR</th>
<th>/GRAPH</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ACTIVITY</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>/DELAYED</td>
<td>I</td>
<td>I</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>/EVERY</td>
<td>W/I</td>
<td>W/I</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>/EXECUTE</td>
<td>I</td>
<td>I</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>/FROM</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>/IMMEDIATE</td>
<td>D</td>
<td>D</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>/TO</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>/UNTIL</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>/WHEN</td>
<td>W/I</td>
<td>W/I</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>/WHILE</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Key**

- yes—the qualifier works
- I—the qualifier is ignored
- W—a warning is given
- D—this qualifier is the default

The following table shows what TRACE format qualifiers will work in conjunction with the four formats.

**Table 5-9  TRACE Format Qualifiers**

<table>
<thead>
<tr>
<th>TRACE</th>
<th>/NORMAL</th>
<th>/ODOMETER</th>
<th>/VECTOR</th>
<th>/GRAPH</th>
</tr>
</thead>
<tbody>
<tr>
<td>/BINARY</td>
<td>yes (w /LOG)</td>
<td>W/I</td>
<td>yes (w /LOG)</td>
<td>W/I</td>
</tr>
<tr>
<td>/COMPRESSED</td>
<td>Z</td>
<td>Z</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>/EXACT</td>
<td>D</td>
<td>D</td>
<td>D (w /HEADER)</td>
<td>D (w /HEADER)</td>
</tr>
<tr>
<td>/FORCED</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>/FULL</td>
<td>yes</td>
<td>yes</td>
<td>yes (w /HEADER)</td>
<td>yes (w /HEADER)</td>
</tr>
<tr>
<td>/HEADER</td>
<td>I</td>
<td>I</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>/LOGNAME</td>
<td>yes</td>
<td>I</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Key**

- yes—the qualifier works
- I—the qualifier is ignored
- W—a warning is given
- D—this qualifier is the default
- Z—this qualifier works only in ZYCAD mode
### Table 5–9 (Cont.)  TRACE Format Qualifiers

<table>
<thead>
<tr>
<th>TRACE</th>
<th>/NORMAL</th>
<th>/ODOMETER</th>
<th>/VECTOR</th>
<th>/GRAPH</th>
</tr>
</thead>
<tbody>
<tr>
<td>/RADIUS</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>/REPLAYABLE</td>
<td>yes (w /LOG)</td>
<td>W/I</td>
<td>yes (w /LOG)</td>
<td>yes (w /LOG)</td>
</tr>
<tr>
<td>/SEPARATOR</td>
<td>I</td>
<td>I</td>
<td>yes (null)</td>
<td>yes (I)²</td>
</tr>
<tr>
<td>/TRACENAME</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>/WIDTH</td>
<td>I</td>
<td>I</td>
<td>yes</td>
<td>I</td>
</tr>
<tr>
<td>/ZEROFILL</td>
<td>I</td>
<td>I</td>
<td>yes</td>
<td>1</td>
</tr>
</tbody>
</table>

¹null is the default separator for /VECTOR
²| is the default separator for /GRAPH

**Key**
- yes—the qualifier works
- I—the qualifier is ignored
- W—a warning is given
- D—this qualifier is the default
- Z—this qualifier works only in ZYCAD mode

---

**Note:** If more customized formatting is required, use the WATCH command in conjunction with PRINT commands instead of TRACE.

A system variable, %ARRAY_ELEMENT, helps trace behavior state arrays. It contains the element number of the last element in an array to change. For example, if you trace A[1] and A[2], and A[2] changes last, %ARRAY_ELEMENT contains a 2.

**ZYCAD Mode**

In the ZYCAD environment, the TRACE command recognizes only the TRACENAME, COMPRESSED, NOCOMPRESSED, and FROM qualifiers. All other qualifiers, if specified, are ignored. The ZYCAD TRACE produces a ZYCAD Output File (ZOF) that is similar to a trace file produced with the DECSIM TRACE command, in the format:

```
TRACE /VECTOR/SEPARATOR= ’ ’ /DELAYED
```

Each TRACE command has a limit of 65K signals.

For more information see:
- PRINT
- WATCH
EXAMPLES

SIM> TRACE A,B,C
Reports transitions in scalar signals A,B,C in /NORMAL format.

SIM> TRACE SUM<3:0>
Reports transitions in a 4-bit port.

SIM> TRACE SUM<0>
Reports transitions in the least significant bit of a 4-bit port.

SIM> TRACE REG<0>
Reports transitions in a 1-bit behavioral state.

SIM> TRACE/LOG=WE0 A,B,C
Sends trace output to log file WEO.TRA.

SIM> TRACE/REPLAYABLE/LOG=GIRAMMA A,B,C
Sends trace output to binary format file GIRAMMA.TRA.

SIM> TRACE/GRAPH /LOG=MULTI A,B,C
Sends trace output in /GRAPH format to MULTI.TRA.

SIM> TRACE/GRAPH/WHEN=(CLK) A,B,C
Reports transitions that occur when CLK is high.

SIM> TRACE/EVERY=100/VECTOR A,B,C
Reports transitions every 100ns in /VECTOR format.

SIM> TRACE/EVERY=[50, 100, 150] A,B,C
Reports transitions at 50, 150, 300, 350, 450, 600ns, etc.

SIM> TRACE/EVERY=(5*G) A,B,C
Reports transitions every 5*G ns where G is nonzero.

SIM> TRACE/FROM=100/TO=1000 A,B,C
Reports transitions that occur between 100ns and 1000ns from current simulation time.
SIM> TRACE/WHILE=(CLK EQL 1) A, B, C

Reports transitions that occur when CLK is high.

SIM> TRACE/FROM=100/TO=1000/WHILE=(CLK EQL 1) A, B, C

Reports transitions that occur between 100ns and 1000ns when CLK is high.

SIM> TRACE/FROM=100/TO=1000/WHEN=(INTR_L) A, B, C

Reports transitions that occur between 100ns and 1000ns when INTR_L is high.

SIM> TRACE/FROM=100/TO=1000/WHILE=(CLK EQL 1)/WHEN=(INTR_L) A, B, C

Reports transitions that occur between 100ns and 1000ns when CLK and INTR_L are high.

SIM> TRACE/BINARY/VECTOR/DELAY/LOGNAME=BTRACE A, B, C

Sends trace output to binary format file BTRACE.TRA, synonymous with /REPLAYABLE.
CANCEL/DISABLE/ENABLE TRACE

The CANCEL TRACE command halts the tracing of the signals specified in the SET TRACE command. The DISABLE/ENABLE TRACE commands temporarily disable and enable the tracing of signals.

**FORMAT**

<table>
<thead>
<tr>
<th>Command</th>
<th>trace_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANCEL TRACE</td>
<td></td>
</tr>
<tr>
<td>DISABLE TRACE</td>
<td></td>
</tr>
<tr>
<td>ENABLE TRACE</td>
<td></td>
</tr>
</tbody>
</table>

**Default Qualifiers**

*None.*

**Optional Qualifiers**

/\ALL
/\ALL=DISABLED
/\ALL=ENABLED

**restrictions**

*None.*

**QUALIFIERS**

/\ALL
Allows you to cancel, enable, or disable all traces with one command.

/\ALL = ENABLED
/\ALL = DISABLED
Used with CANCEL TRACE, allows you to cancel all enabled traces or all disabled traces. This qualifier is useful when you have many traces and want to cancel all but a few. You disable the traces you want, cancel the rest (with CANCEL TRACE/\ALL = ENABLED), then enable the traces you want.

**DESCRIPTION**

The CANCEL TRACE command stops performing the trace; in order to continue the trace, the TRACE command must be reissued.

The DISABLE TRACE command temporarily stops sending output to the trace file until it is enabled again.
EXAMPLE

SIM> CANCEL TRACE tracel

**ZYCAD Mode**

Just as in soft DECSIM, CANCEL TRACE and its qualifiers work in ZYCAD mode.
SHOW TRACE

The SHOW TRACE command shows the current trace name(s) and the qualifiers you have selected.

FORMAT

SHOW TRACE

Default Qualifiers
None.

Optional Qualifiers
/FULL

restrictions
None.

QUALIFIERS

/FULL
Displays the list of signals for each TRACE command.

DESCRIPTION

The SHOW TRACE command prints the names of all currently established traces (disabled or enabled). It will also show the qualifiers selected, as well as some of the defaults.

ZYCAD Mode
SHOW TRACE also works in ZYCAD mode.
TYPE (Command Macro)

The TYPE command is similar to the TYPE DCL command. It displays a file on your terminal without leaving the DECSIM simulator or losing the network.

**FORMAT**

```
TYPE filename
```

**Default Qualifiers**

```
None.
```

**Optional Qualifiers**

```
None.
```

**restrictions**

- A file must be closed or checkpointed before it can be read by the DECSIM TYPE commands.

**DESCRIPTION**

This command allows you to see a file without exiting DECSIM and disturbing the DECSIM image. It is especially useful for examining TRACE or LOG files that have just been written, or indirect files about to be executed. The default extension is .IND.

**EXAMPLE**

```
SIM> TYPE TEST1.LOG

<file is printed here...>
```

**TYPE Macro Definition**

```
SET MACRO /COMMAND TYPE -
[ $FILENAME ] =
PRINT %INFILE( $FILENAME );
SENDMACRO
```
UNTIL

See WHILE.
USE

Loads a compiled network description file and readies it for simulation.

**FORMAT**

**USE**  [/qualifier] filename

Default Qualifiers
None.

Optional Qualifiers
/EXECUTABLE
/INPUT_PARTITION=file_specification network_object_file
/OUTPUT_PARTITION=file_specification network_object_file

**ARGUMENTS**

filename
Specifies the file from which to read the network.

**QUALIFIERS**

/EXECUTABLE=filename
Specifies an alternate name for the .EXE file produced when compiling a behavior model. Default is the same name as the .NET file.

/INPUT_PARTITION=file_specification network_object_file
A ZYCAD-specific qualifier pointing to a file containing instructions on how to segment, or partition, the model network. This allows you to specify what parts of your model are simulated in which ZYCAD S-module, and can speed up simulation of large networks. See Appendix E for information on the input partition file.

/OUTPUT_PARTITION=file_specification network_object_file
A ZYCAD-specific qualifier that allows you to control the file specification of the partition output file (PAO). If this qualifier is not specified, the PAO is generated using your current directory, the file name of the partition input (PAI), and the extension PAO. If this qualifier is not specified and there is no PAI file, the PAO is generated using your current directory, the file name of your network, and the extension PAO. See Appendix E for information on the output partition file.
USE

DESCRIPTION
The USE command performs half the functions of the COMPILE command. It loads a compiled network description. If you try to load a network that has already been loaded, USE returns the memory allocation used by the old network, resets statistics monitoring, and reads in the new network. In addition, all currently defined WATCH's and TRACE's are canceled, and the SCOPE setting returned to top level. The time resolution and default time units are also set and displayed. However, MACRO definitions and STATE variables are unaffected.

There are two files associated with behavior models that you must have access to in order to USE a behavioral model. They are the .NOB and the .EXE files. These are created when you compile your model.

Zycad Mode
The USE command allows you to load a network that has been precompiled with the default library of ZGATES. There is no error checking to ensure that the proper library was used; however, DECSIM in Zycad mode won't successfully build a network if it doesn't recognize a model. The Zycad network is built at the end of the loading process.

For more information see:

| COMPILe | SAVe |
| RESTORE | Zycad |

EXAMPLE

SIM> USE MOS1
  Reading File: DISK$DCB1:[SHERWOOD] MOS1.NOBU12
  Time Resolution = 1ns, default unit is nS

SIM>

The network is now loaded.
VAXSTATISTICS (Command Macro)

Allows you to capture the output of the DCL command SHOW PROCESS/ACCOUNTING/QUOTAS/PRIVILEGES in a log file, without leaving the DECSIM simulator and losing your simulation session.

FORMAT

VAXSTATISTICS

Default Qualifiers
None.

Optional Qualifiers
None.

restrictions
None.

DESCRIPTION

The VAXSTATISTICS command creates a log file called DECSIMPROCESS.STA in the parent process area. Then it creates a sub-process in which it calls the VAXSTATS.COM file from the DECSIM area on your system. This indirect command file writes the output of the DCL command SHOW PROCESS/ACCOUNTING/QUOTAS/PRIVILEGES to the log file.

The command file prompts you for an optional text string in case you intend to take multiple "snapshots" of the process and need to identify at what point in the simulation session each snapshot was taken. Note that each time you execute this command, a new version of DECSIMPROCESS.STA is opened.

EXAMPLE

SIM> sho macro 'vaxstatistics'
  SET MACRO /COMMAND /ABBREV:2 VAXSTATISTICS =
  BEGIN
  LOG/LOGN:statfaclog decsimprocess.sta;
  CANCEL LOG statfaclog;
  VMS @decsimsys:vaxstats.com;
  END SENDMACRO;
SIM> type decsimsys:vaxstats.com
VAXSTATISTICS (Command Macro)

$! File: VAXSTATS.COM
$!****** Initialize *****!
$ VER := 'F$VERIFY (0)'
$ SET CONTROL=(T,Y)
$ ON ERROR THEN GOTO exit
$ PID = f$getjpi ("", "owner")
$ terminal = f$getjpi ("", "prcnam")
$****** OPEN Input / Output FILES *****!
$ OPEN/WRITE tty 'f$getjpi (PID,"terminal")'
$ OPEN/APPEND/ERROR = exit statlog DECSIMPROCESS.STA
$ WRITE tty "Enter a 100 charater (or less) STAT-RUN IDENTIFIER if you wish: "
$ READ tty id /prompt=""
$ WRITE statlog "=================================================================================="
$ WRITE statlog " 'id'"
$ WRITE statlog "=================================================================================="
$ SHOW PROC/ALL/ID:'PID'/OUTPUT:statlog
$ CLOSE statlog
$ WRITE tty "Simulation-process statistics written (file: DECSIMPROCESS.STA)."
$ GOTO exit
$ABORT:
$EXIT:
$ CLOSE tty
$ DEFINE/USER SYSS OUTPUT NLAO:
$ DEFINE/USER SYSS ERROR NLAO:
$ ver = f$verify(ver)

SIM> vax
\I-CL0, Closed: DOC$DISK:[SAWA.DEC$IM.HAZARDS]DECSIMPROCESS.STA;2

Enter a 100 charater (or less) STAT-RUN IDENTIFIER if you wish:
$snapshot 1
Simulation-process statistics written (file: DECSIMPROCESS.STA).

SIM> type decsimprocess.stat

DEC SIM  Version V5.4-3801  6-FEB-1989  11:08:06

LOG file closed 6-FEB-1989 11:08:06

--------------------------------------------------------------------------------
Snapshot 1
--------------------------------------------------------------------------------

Fid: 3040212C Proc. name: Friscilla UIC: [DEC$IM, SAWA]
Priority: 4 Default file spec: Not available

Devices allocated: $4$VTA316: (ELSIE)

Process Quotas:
Account name:
CPU limit:  Infinite Direct I/O limit: 18
Buffered I/O byte count quota: 19168 Buffered I/O limit: 18
Timer queue entry quota: 9 Open file quota: 95
Paging file quota: 23094 Subprocess quota: 3
Default page fault cluster: 64 AST quota: 19
Enqueue quota: 500 Shared file limit: 0
Max detached processes: 0 Max active jobs: 0

Accounting information:
Buffered I/O count: 4502 Peak working set size: 2614
Direct I/O count: 983 Peak virtual size: 10885
Page faults: 13219 Mounted volumes: 0
Images activated: 24
Elapsed CPU time: 00:00:23.08
Connect time: 01:35:45.16

5-234
VAXSTATISTICS (Command Macro)

Process privileges:

TMPMBX  may create temporary mailbox
NETMBX  may create network device
VMS_COMMAND (Command Macro)

Spawns a subprocess and puts you at the VMS DCL level. This is a command macro that expands to SPAWN/WAIT. (See the SPAWN command).

**FORMAT**

VMS_COMMAND \[ DCL–command–string \]

**Default Qualifiers**

*None.*

**Optional Qualifiers**

*None.*

**restrictions**

- The VMS_COMMAND without a parameter will not work from an indirect file or a command file. If you want to execute more than one DCL command from an indirect file or a command file, list the DCL commands in an indirect file and invoke this file with the VMS_COMMAND.

  For example: `SIM> VMS_COMMAND @test` (where the file test contains a list of DCL commands).

- When executing a DCL command, a CTRL/C interrupt will take you to the CTRL–C> prompt. Do not use the "A" (abort) response to this prompt. If you use the abort response, further DCL command processing will not work correctly. CTRL/Y should also not be used because this aborts the current operation as well as your DECSIM simulation session.

- DCL commands that take input from the terminal may not be included on the same line as VMS_COMMAND. For these cases, use the simple form of the VMS_COMMAND (with no argument) and then specify the DCL command on the line following the VMS_COMMAND. Use the following format.

  `SIM> VMS
$ MAIL mail text exit
$ LOGOUT
SIM>`

- DCL commands that set the characteristics of an input or output device may not be included on the same line as the VMS_COMMAND. For these cases, specify the DCL command on the following line as shown above.

- If the single–line method is used, DECSIM lexical operators "+", "–", and ",,” take precedence and have DECSIM meanings, not DCL meanings.

  For example:

  `SIM> VMS dir *.net; sh ti`

  "Sh ti" will be the simulation clock time (from DECSIM) not the date and time (from DCL).
ARGUMENTS  

**DCL-command-string**
Indicates a specific DCL command to be executed. The command is executed, and control is returned to a DECSIM prompt.

DESCRIPTION

The VMS_COMMAND directs the VAX/VMS DCL command interpreter to execute your command as a subprocess. Once you invoke VAX/VMS DCL, all your VMS logical name and symbol definitions are in effect. Your default directory is the same as your default directory when you invoked DECSIM (or CHAS if you are running under CHAS).

When the VMS_COMMAND is invoked without a parameter, VAX/VMS DCL displays the dollar sign prompt ($) to indicate that it is ready to accept input. You can specify any DCL command in response to the prompt. All DCL commands are fully described in the VAX/VMS Command Language User's Guide.

When you want to return to the DECSIM command level, specify the DCL command LOGOUT. The LOGOUT command logs you out of the subprocess.

When invoked with a parameter (a DCL command), the DCL command is executed and, on completion of the command, control is returned to DECSIM.

For more information see:

EXIT
QUIT

EXAMPLE

```plaintext
SIM> VMS_COMMAND
$ SHOW DEFAULT
  LVSS:[RANDALL.WORKDECSIM]
$ 1any DCL commands
$ LOGOUT
SIM>
SIM> VMS_COMMAND DIR *.net
Directory LVSS:[RANDALL.WORKDECSIM]
BADD1.NET;4  1BEHAVIOR.NET;3  COUNTER.NET;16  COUNTER1.NET;2
MOS.NET;2   MOOT.NET;5    NOT.NET;2      SIMPLE.NET;3
TADDER.NET;5 TEST.NET;1

Total of 10 files.
SIM>
```
WATCH

Sets up a watch_condition on a signal, register, memory, or internal state value transition or a behavior statement execution. When the watch_condition is satisfied, the WATCH action is executed. You can include most DECSIM commands in the action. After execution of the action commands, simulation continues or you are returned to interactive command mode.

WATCH can be used with vector items such as behavior and memory ports. It can also be used with behavior states and arrays, both global and local. You cannot watch a memory primitive array.

The TRACE command is a special form of the WATCH command; its operation is described in a separate section.

CANCEL/DISABLE/ENABLE WATCH and SHOW WATCH are described as separate commands, following this description of SET WATCH.

FORMAT

[SET] WATCH [ /qualifier... ] [ access_list ] DO [ statement ]

Default Qualifiers
/Delayed
/From=0
/To=Infinity

Optional Qualifiers
/Call=(routine_name_list)
/Every=time
/Every=(time_list)
/Every=(time_expression)
/execute
/From=\[ ]time[units]
/Immediate
/Modify
/Once
/Resume=(access_list)
/Return=(access_list)
/Statement: [filename]: [line_number]
/To=\[ ]time[units]
/Until=(expression)
/Wait=(access_list)
/Watchname=watch_name
/When=(expression)
/While=(expression)

restrictions

- The WATCH command cannot be executed until you have a loaded network.

5–238
- If a watch_action calls an indirect file, the file is read at the time the watch_action is defined. If the file is changed, the watch_action must be redefined.

**ARGUMENTS**

**access_list**

Access_list is a list of names separated by commas.

*Note:* A change in a system variable will not cause the evaluation of an expression in a watch_expression. You can use system variables in expressions, but not in a watch access list.

**statement**

If no statement is specified, the default is DO HALT.

**QUALIFIERS**

All the qualifiers must precede the access list.

//CALL=(routine_name_list)

Activates the watch_action whenever the specified behavior routine is entered, whether it is called from the command language, or activated as a result of the ACTIVATE command or a port entry. This qualifier activates the watch_action after the actual parameters are copied, but before local states are initialized. The default watch_action when you include this qualifier is to print the routine name.

//DELAYED

Specifies that the watch_action commands are performed at the end of the time slot. This qualifier is applicable for watches triggered by signal or state transitions. When applied to local states, //DELAYED is interpreted as //IMMEDIATE, and no warning is given.

//EVERY=time

//EVERY=[time_list]

//EVERY=(time_expression)

Indicates that the watch is triggered by the simulator clock. Times can be absolute or relative. If a time list contains one or more absolute times, all times in the list are considered to be absolute. Otherwise, times are relative. A time list consists of times separated by commas. The list is enclosed in square brackets. A time expression may also be used. Enclose the expression in parentheses (for example, //EVERY=(5*A)). Times must be given in chronological order with no 0 relative time or repeated absolute time. Triggering is repeated for a relative time list. //IMMEDIATE is not applicable with this qualifier.

//EXECUTE

Causes immediate execution of a watch_action from the command language at the time of the WATCH/EXECUTE. This can be used for debugging the watch_action or for triggering the WATCH from command mode or an indirect file or another watch_action.

With the //EXECUTE qualifier, you can make a subroutine facility in the command language. See note under behavior WATCH qualifiers.
/FROM=[ ]time[units]
/TO=[ ]time[units]
/FROM and /TO specify starting and stopping times for the WATCH command. The specified time must be greater than or equal to 0 and also greater than or equal to the current simulator clock. The /TO time must be greater than or equal to the /FROM time. When the simulator clock reaches the /TO time, the WATCH is canceled.

/IMMEDIATE
Executes the watch_action right after the transition matures. For watches in a ported model, the watch_action will not be executed until the end of the time slot unless /IMMEDIATE is used. This delayed action is different than that for a nonported model, where watch_actions are always executed immediately.

/MODIFY
Allows modification of an existing WATCH command. Qualifiers that may be modified are /ONCE and /EXECUTE. If you supply a new DO action, the watch_action commands are replaced.

Note: /MODIFY must be used in conjunction with the /WATCHNAME qualifier. When used with /MODIFY, /WATCHNAME merely refers to an already existing watch name, indicating the WATCH command you will modify.

/ONCE
Causes a CANCEL WATCH after the first successful trip_condition test. This is equivalent to a CANCEL WATCH inside the DO action.

/RESUME=(access_list)
Allows you to see when a routine suspended by a WAIT command has resumed executing. The default watch_action is to print the routine name.

/RETURN=(access_list)
/RETURN activates the watch_action when the specified behavior routine exits. Any return value in the routine has been updated in a local state in the calling routine, a global state in a model, and always in the system variable %ROUTINEVALUE. (Note that %ROUTINEVALUE system variable is accessible only from the command language and not from behavior models.) The default watch_action when you include this qualifier is to print the routine name.

/STATEMENT:[filename] line_number
Activates the watch_action when the specified behavior statement is about to be executed. To use this qualifier, your behavior model must have been compiled with COMBEH filename /DEBUG or with COMPILE/BEHAVIOR/DEB. The /LIS qualifier should also be used to produce a .LIS file. Your top-level model must have been compiled with COMPILE/DEBUG. The line_number refers to the left-most column in the .LIS file. The optional filename refers to the name of the model or module. Do not include the file extension or other elements of the full file specification.
/UNTIL=(expression)
The expression specified by /UNTIL is tested whenever one of its components changes. If the value is true, the watch is canceled. The expression evaluation occurs at the end of the time slot and is not affected by the /IMMEDIATE qualifier.

/WAIT=(access_list)
Activates the watch_action when a WAIT statement suspends execution in the specified "watched" routine. The default watch_action is to print the routine name.

/WATCHNAME= watch_name
Provides a name for the WATCH statement for use in CANCEL and DISABLE/ENABLE WATCH, as well as for the /MODIFY and /EXECUTE qualifiers. If omitted, a default unique name is supplied. SHOW WATCH shows the watch_name.

/WHEN=(expression)
Triggers the watch every time the /WHEN expression becomes true. Even though the expression operands may change, the watch is triggered only when the expression result changes and that result (the new value) is true.

/WHILE=(expression)
The expression specified by /WHILE enables watching when true, and disables watching when false. The expression evaluation occurs at the end of the time slot and is not affected by the /IMMEDIATE qualifier.

DESCRIPTION
The WATCH command monitors the activity of network values, behavior model statements, and internal states during simulation. [SET] WATCH establishes the watch condition(s) and action(s) that will be taken. Whenever the requested test watch_condition is satisfied, DECSIM executes the specified list of commands. This list of commands is called the action of the command. If an action is not specified, simulation stops and control returns to the user's terminal or currently active indirect file.

The action commands are stored internally for later execution. They are executed every time the WATCH trip_condition occurs (as long as the WATCH is still in effect – see DISABLE/ENABLE WATCH). The commands are parsed when typed, so escape prompting and syntax error reporting are in effect during command entry. If an error occurs during the execution of the watch_action, DECSIM completes as much as possible and then continues with the next command. For example, if an EXAMINE command included in the watch_action tries to examine a signal that does not exist, an error message is reported and DECSIM proceeds to the next command in the watch_action.

Macros in the watch_action command list are expanded when the WATCH command is entered. Thus the macro expansion remains, even if the macro definition is modified or deleted. A macro expansion listing report occurs only when the watch_action is typed in, not each time the trip occurs and the watch_action is executed.

The command list action may contain <CR>'s, semicolons, and other WATCH commands. The only restriction is that unmatched parentheses in signal names must be enclosed in double quotation marks.

5-241
Any command language expression may be used in the WATCH /WHEN/WHILE/UNTIL expression. If the expressions contain word or bit subscripts, they are evaluated only once—when the SET WATCH command is entered.

Three ways to activate the WATCH action are:

1. Immediately from command mode or from another watch-action.

   The [SET] WATCH/EXECUTE command executes the watch_action at the time the command is entered. This execution is useful for debugging the WATCH definition. It is sometimes used instead of a macro for executing a group of commands. It is not triggered by time or simulation events.

2. From simulation when a signal transition or event, such as a call to a routine, occurs.

   A change in any signal in a WATCH access list triggers the watch_action. /DELAYED causes an action to be executed once, despite multiple signal changes within a time slot. /IMMEDIATE causes an action to be executed every time a signal changes, even if the signals all change within the same time slot.

   A change in a /WHEN expression triggers the watch_action. DECSIM evaluates the /WHEN expression every time a signal operand in that expression changes. If the result of the expression is true, the action is executed. Only one expression may be given; if you want more, connect the expressions with the OR operator, or use an access list format and place them in IF statements within the watch_action.

3. From the simulation time.

   The /EVERY qualifier causes the WATCH to occur at the specified times.

The following commands may not be used within a watch_action:

COMPILE     SAVE     USE
RESTORE     SIMULATE

Note: SET INDIRECT (©) may be used in a watch_action. However, indirect files are stored, so that text substitution occurs at watch definition time. The indirect file could be deleted, but the commands would still be executed. Be sure to enclose multiple commands in a BEGIN-END block.

If a WATCH name is not supplied, DECSIM will inform the user of the default name selected by the system.

The HALT command used within a watch_action causes the simulator to return to command mode immediately and leave the rest of the watch_action unexecuted. The return to command mode happens when the WATCH mechanism returns control to the simulation scheduler.


Note: In the example below, false triggering (or no triggering) can occur if A_REG and B_REG change in the same time slot.
WATCH/IMMEDIATE/WHEN=(A_REG EQL B_REG) DO cmd

<table>
<thead>
<tr>
<th>time</th>
<th>A_REG</th>
<th>B_REG</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**** False trigger ****

For more information see:
- CANCEL WATCH
- ENABLE WATCH
- HALT
- DISABLE WATCH
- SHOW WATCH
- TRACE

---

**EXAMPLES**

SIM> WATCH/EVERY=[100,75] DO PRINT \$TIME, \$PC=\$/PC<15:0>/RADIX:16
SIM> WATCH/EVERY=(5*A) DO PRINT \$TIME
SIM> WATCH bus_interrupt DO
MORE> BEGIN
MORE> PRINT 'Whoops!'
MORE> HALT
MORE> END
SIM> WATCH/STATEMENT:model_name
SIM> WATCH/WHEN=(A1 AND A2) DO PRINT \$TIME

---

**Control Flow Tracing**

This tracing is shown after you enter the "DEBUG/LOOP" command.

[ Start WATCH action <name> at <time> ]
[ End of WATCH action <name> ]
CANCEL/DISABLE/ENABLE WATCH

The CANCEL WATCH command terminates the watching conditions and deletes them from DECSIM. The DISABLE/ENABLE WATCH commands temporarily disable and enable the watch conditions.

**FORMAT**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANCEL WATCH</td>
<td><code>watch_name</code></td>
</tr>
<tr>
<td>DISABLE WATCH</td>
<td><code>watch_name</code></td>
</tr>
<tr>
<td>ENABLE WATCH</td>
<td><code>watch_name</code></td>
</tr>
</tbody>
</table>

**Default Qualifiers**

None.

**Optional Qualifiers**

/ALL
/ALL=DISABLED
/ALL=ENABLED

**restrictions**

- A WATCH/EXECUTE/MODIFY/WATCHNAME=xx command is still executable even after DISABLE WATCH xx has been executed.

**QUALIFIERS**

/ALL

Allows you to cancel, disable, or enable all watches with one command.

/ALL = DISABLED
/ALL = ENABLED

Used with CANCEL WATCH, allows you to cancel all enabled watches or all disabled watches. This qualifier is useful when you have many watches and want to cancel all but a few. You disable the watches you want, cancel the rest (with CANCEL WATCH/ALL = ENABLED), then enable the watches you want.

**DESCRIPTION**

The CANCEL WATCH deletes a watch condition and action as established in a SET WATCH command.

DISABLE WATCH forces the watch condition to be false (never taking the watch action), until a subsequent ENABLE WATCH is given.
EXAMPLE

SIM> DISABLE WATCH nightwatch

ZYCAD Mode

CANCEL, ENABLE, and DISABLE WATCH are valid for both ZYCAD and soft DECSIM.
SHOW WATCH

The SHOW WATCH command displays the current WATCH names as established in the SET WATCH command.

FORMAT

SHOW WATCH

Default Qualifiers
None.

Optional Qualifiers
/FULL

restrictions
None.

QUALIFIERS

/FULL
Displays the list of signal names for each WATCH command.

DESCRIPTION

SHOW WATCH prints the names of all currently established watches and whether they are enabled or disabled. WATCH and TRACE statistics are also printed. These statistics are reset each time a new network is loaded.

EXAMPLES

1 SIM> SHOW WATCH
WATCH2
WATCH1
Watch/Trace Statistics
Transitions WATCHEd 5
Expressions evaluated 5
WATCH actions executed 3
TRACE actions executed 0
SIM> SHOW WATCH/FULL
WATCH2
  A
  B
  C
  D
WATCH1
  G
Watch/Trace Statistics
  Transitions WATCHEd   5
  Expressions evaluated  5
  WATCH actions executed 3
  TRACE actions executed 0
WHILE and UNTIL

The WHILE and UNTIL commands provide iterative command execution.

**FORMAT**

[label:] WHILE ( test_condition ) DO statement_or_block
[label:] UNTIL ( test_condition ) DO statement_or_block

Default Qualifiers
None.

Optional Qualifiers
None.

restrictions
None.

**DESCRIPTION**

The WHILE command is used to iterate a command sequence as long as a given test_condition is true. The UNTIL command iterates the command sequence as long as the test_condition is false. The test_condition is evaluated at the beginning of each loop iteration.

There is no form of DO stmt WHILE expr in the command language.

For more information see: FOR IF LEAVE

**EXAMPLE**

SIM> WHILE (RESET NEQ 1) DO
MORE>   BEGIN
MORE>   D CLR=0; SIM 100
MORE>   D CLR=1; SIM 100
MORE>   END

Control Flow Tracing

The following tracing is shown when the “DEBUG /LOOP” command is in effect.
WHILE and UNTIL

SIM> def reg<7:0>
SIM> ev reg=100#2
    [ EVALUATE assignment: reg = 100#2 ]
SIM> while reg neq 0 do
    MORE> begin
    MORE>     print reg/radix:2/wid:3
    MORE>     ev reg=reg sr0 1
    MORE>     end
    [ WHILE condition TRUE ]
    [ Entering block level 1 ]
100
    [ EVALUATE assignment: REG = 10#2 ]
    [ Leaving block level 1 ]
    [ WHILE condition TRUE ]
    [ Entering block level 1 ]
10
    [ EVALUATE assignment: REG = 1#2 ]
    [ Leaving block level 1 ]
    [ WHILE condition TRUE ]
    [ Entering block level 1 ]
1
    [ EVALUATE assignment: REG = 0#2 ]
    [ Leaving block level 1 ]
    [ WHILE condition FALSE ]
SIM>
The ZYCAD command establishes the context for ZYCAD simulation and determines the accelerator to be used.

**FORMAT**

[SET] ZYCAD  
CANCEL ZYCAD  
DISABLE ZYCAD  
ENABLE ZYCAD  
SHOW ZYCAD

**Default Qualifiers**

/LE_SIMULATION  
/ZSTATE

**Optional Qualifiers**

/FE_SIMULATION  
/NOZSTATE  
/REPLAY=filename

**restrictions**

None.

**QUALIFIERS**

/FE_SIMULATION

Specifies that the ZYCAD simulation be performed on the ZYCAD fault evaluator (ZYCAD$FE_NAME), which does good simulation and concurrent fault simulation. CANCEL ZYCAD/FE_SIMULATION causes the simulation to be performed in the default mode, /LE.

/LE_SIMULATION

Specifies that the ZYCAD simulation be performed on the ZYCAD logic evaluator (ZYCAD$LE_NAME), which does good simulation and serial fault simulation. This is the default.

/REPLAY=filename

Uses the .ZBR file produced by SIMULATE/REPLAY to replay a hard ZYCAD simulation using soft DECSIM. You do not need to specify a filename unless you have used the SIMULATE/RUNNAME command. When you replay a simulation more than once, the RESTORE command must be used before each replay to reset the clock and the state of the network. If no time is specified, the entire .ZBR file is replayed.
The following commands are valid when replaying a simulation:

BREAK  RESTORE  TIME
DEBUG  SAVE  TIMING
EVALUATE  SIMULATE  TRACE
EXAMINE  STATISTICS  WATCH
LOG

Normally, WATCH and TRACE output of ZYCAD simulation is displayed as TRACE/VECTOR in the .ZOF file. However, when you use WATCH and TRACE commands prior to using SIMULATE/REPLAY, enough information is captured during the ZYCAD simulation so that when you replay the ZYCAD simulation in soft DECSIM, the WATCH and TRACE commands have the normal full functionality.

You cannot modify signal states in a replayed simulation; you can only watch. Thus, the following commands are invalid when replaying a simulation:

CONTEXT  FAULT
DEPOSIT  LOAD
DETECT  SAVE
DROP  PATTERN

COMPILEx and USE close the .ZBR file being replayed and return DECSIM memory to its initial state.

The .ZBR file produced by SIMULATE/REPLAY is replayed in the following context:

1  RESTORE file
2  SET ZYCAD/REPLAY=file.ZBR
3  Cancel unwanted TRACE commands
4  Apply TRACE and WATCH commands
5  SIMULATE

/REPLAY is used with soft DECSIM and thus cannot be used with /LE_SIMULATION or /FE_SIMULATION, which specify hard ZYCAD accelerators.

System variables produced by /REPLAY, which can be displayed with EVALUATE or PRINT, are described in Section 4.2.6.3.

/ZSTATE
Enables the reporting of the Z state. This is the default. Soft DECSIM and DECSIM/ZYCAD produce identical results when reporting a Z.

/NOZSTATE
Disables the reporting of the Z state.
DESCRIPTION

The ZYCAD Logic and Fault Evaluator is a batch hardware simulation engine that executes approximately one million events per second when the ZYCAD pipeline is kept full. It is primarily intended as a performance improvement tool for DECSIM. Types of simulations for which ZYCAD is useful include design verification, regression testing, and, in some cases, fault simulation.

To use the ZYCAD unit, compile your network and set up test conditions in ZYCAD mode as you would for interactive DECSIM. When you type SIMULATE, your network is sent via DECnet to the system where the ZYCAD unit resides. Your network is then simulated and the results sent back to you.

The SET ZYCAD command creates the ZYCAD environment. All future commands apply only to the upcoming ZYCAD run.

The CANCEL ZYCAD command terminates a ZYCAD session and deallocates all data structures that represent the ZYCAD network and its stimuli, such as uses of the PATTERN, LOAD, TRACE, FAULT, and DETECT commands.

SHOW ZYCAD displays the current ZYCAD mode (LE or FE.)

The following DECSIM commands have been modified to create data structures for the ZYCAD input (ZIF) file.

<table>
<thead>
<tr>
<th>Command</th>
<th>Original Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPILE</td>
<td>PATTERN</td>
</tr>
<tr>
<td>DETECT</td>
<td>SIMULATE</td>
</tr>
<tr>
<td>FAULT</td>
<td>TRACE</td>
</tr>
<tr>
<td>LOAD/MICRO</td>
<td>USE</td>
</tr>
</tbody>
</table>

There is no change in the syntax or function of these commands. However, some commands, such as DETECT and SIMULATE, have ZYCAD–specific qualifiers.

For more information see:

<table>
<thead>
<tr>
<th>Command</th>
<th>Original Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPILE</td>
<td>DETECT LOAD SIMULATE USE</td>
</tr>
<tr>
<td>DEPOSIT</td>
<td>FAULT PATTERN TRACE WATCH</td>
</tr>
</tbody>
</table>
EXAMPLE

SIM> COMPIL; ADDMEM
   Reading File: SNAP$:[ZYCAD.TESTS]ADDMEM.NOB;10
   Time Resolution = 1 nS, default unit is nS
   Creating ZYCAD Network...
   Number of ZYCAD Gates Used: 24
   ZYCAD Network Successfully Built

SIM> LOAD DATA
   Loading state: DATA
   7 States Loaded.

SIM> TRACE A3,A2,A1,A0,B3,B2,B1,B0,CIN,COUT,S3,S2,S1,S0
SIM> PATTERN/FILE:ADDMEM/ABSOLUTE 4TIME,READ,ADDR2,ADDR1,ADDR0
   %1, [SIMPNI] The pattern name is PATTERN2
SIM> SIM 1200
   Submitting simulation on remote node...

SIM> EXIT
Command and Variant Prefix Summary

The following table lists all the DECSIM commands and their prefixes. For each command, the valid prefixes are indicated with an "x." If you do not specify a prefix, DECSIM assumes the default prefix.

Although the DECSIM–supplied macros do not use prefixes, they are included in the following table.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Default Prefix</th>
<th>SET</th>
<th>SHOW</th>
<th>CANCEL</th>
<th>DISABLE</th>
<th>ENABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVATE</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEGIN-END</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BREAK</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>CALL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMBEH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPILE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONFIGURE</td>
<td>SET</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTEXT</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTINUE (macro)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEBUG</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEPOSIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DETECT</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>DROP</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDITCOMMANDMACRO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDITTEXTMACRO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERROR</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVALUATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXAMINE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FAULT</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HALT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HAZARD</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>HELP</td>
<td>SHOW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keyword</td>
<td>Default Prefix</td>
<td>SET</td>
<td>SHOW</td>
<td>CANCEL</td>
<td>DISABLE</td>
<td>ENABLE</td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
<td>-----</td>
<td>------</td>
<td>--------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>IF</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>INDIRECT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INFORM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEY</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LABELS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEAVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>MACRO</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOS</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PATTERN</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>PAUSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRINT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QUIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RADIX</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RECALL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REPLAY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESTORE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESUME</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAVE</td>
<td>SET</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCOPE</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SELECT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHARE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIMULATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPAWN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STATE</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STATISTICS</td>
<td>SHOW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STEP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYMBOLS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TERMINAL</td>
<td>SET</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>SHOW</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIMING</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRACE</td>
<td>SET</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>TYPE (macro)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNTIL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>USE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A–2
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Default Prefix</th>
<th>SET</th>
<th>SHOW</th>
<th>CANCEL</th>
<th>DISABLE</th>
<th>ENABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMS_COMMAND</td>
<td>SET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WATCH</td>
<td>SET x x x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WHILE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZYCAD</td>
<td>SET x x x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ASCII Character Definitions for DECSIM Languages

The following tables list ASCII characters and their use in the three DECSIM languages: SX, NETPRO, and the DECSIM command language.

Table B–1 lists DECSIM control characters; Table B–2 lists the printing ASCII characters and their DECSIM definitions.

The following table defines the 4 control characters used in the DECSIM command language. Except for those 4, the control characters (ASCII 0–31) have no special significance to DECSIM.

**Table B–1 Nonprinting Control Characters**

<table>
<thead>
<tr>
<th>Character</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL/C</td>
<td>Emergency interrupt.</td>
</tr>
<tr>
<td>CTRL/G</td>
<td>Erases current command being entered and returns to the top–level prompt.</td>
</tr>
<tr>
<td>CTRL/Z</td>
<td>When entered at the SIM prompt, exits from DECSIM. When entered with LOAD/FILE=SYS$INPUT or PATTERN/FILE=SYS$INPUT, signifies end–of–file, completing the execution of PATTERN or LOAD, allowing simulation to continue with the next command or the top–level prompt. When entered in HELP, provides a quick exit to the SIM&gt; prompt.</td>
</tr>
<tr>
<td>TAB</td>
<td>Provides escape, or tab, prompting.</td>
</tr>
</tbody>
</table>

**Table B–2 Printing Characters**

<table>
<thead>
<tr>
<th>Character</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>Keyword separator.</td>
</tr>
<tr>
<td>!</td>
<td>Comment character.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Quote character that allows you to include nonstandard characters in a signal name by enclosing them in quotation marks. For example, &quot;A+B&quot;. Two consecutive quote characters in a quoted name cause one quote character to be displayed. For example, &quot;A&quot;&quot;B&quot; is displayed as A*B.</td>
</tr>
<tr>
<td>#</td>
<td>Radix specifier.</td>
</tr>
<tr>
<td>$</td>
<td>Macro facility character for parameters and concatenation; valid character for file names.</td>
</tr>
<tr>
<td>%</td>
<td>Valid signal name character; indicates system variable if used as the first character in a name.</td>
</tr>
</tbody>
</table>
### Table B-2 (Cont.) Printing Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Concatenates values, which can be specified explicitly or by signal name.</td>
</tr>
<tr>
<td>'</td>
<td>Apostrophe: Encloses text strings, as well as macro and file names containing nonstandard characters; prevents macro expansion.</td>
</tr>
<tr>
<td>(</td>
<td>Encloses expressions to indicate precedence; encloses behavior parameters; indicates behavior routine; encloses command language qualifier argument list.</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk: Wildcard for signal names (even when enclosed in quotation marks); multiplication operator in expressions.</td>
</tr>
<tr>
<td>+</td>
<td>Addition operator.</td>
</tr>
<tr>
<td>,</td>
<td>Comma: A list separator.</td>
</tr>
<tr>
<td>-</td>
<td>Dash: A subtract/negative operator in expressions; denotes line continuation in the command language; assertion in signal names.</td>
</tr>
<tr>
<td>.</td>
<td>Period: A file extension separator; optional file revision separator; hierarchical signal name separator (even when enclosed in quotation marks); denotes fractional time values.</td>
</tr>
<tr>
<td>/</td>
<td>Indicates a qualifier to a command; in an expression, is the divide operator; separates pin name from signal name in a name-based instance statement.</td>
</tr>
<tr>
<td>0-1</td>
<td>Signal states.</td>
</tr>
<tr>
<td>0-9</td>
<td>Integers.</td>
</tr>
<tr>
<td>A-F</td>
<td>Hex integers.</td>
</tr>
<tr>
<td>A-Z</td>
<td>Uppercase letters.</td>
</tr>
<tr>
<td>a-z</td>
<td>Lowercase letters, which are treated as uppercase except when located in strings.</td>
</tr>
<tr>
<td>H,L</td>
<td>Followed by #2, indicate the 1 and 0 states.</td>
</tr>
<tr>
<td>Z,U</td>
<td>Followed by #2, indicate the high impedance and undefined states.</td>
</tr>
<tr>
<td>:</td>
<td>Subscript range separator; label definition; optional connector of arguments to command language qualifiers; separates min, nom, and max parameter values.</td>
</tr>
<tr>
<td>;</td>
<td>Statement separator; used in file names that include revision number; separates parameters in the MACRO call syntax.</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>Bit subscripts for names and expressions.</td>
</tr>
<tr>
<td>=</td>
<td>Assignment operator in behavior models, as well as in command language DEPOSIT and EVALUATE commands; separates outputs from inputs in network connection lists; connects arguments to command language qualifiers; joins model and macro parameter default values to the formal parameter in NETPRO.</td>
</tr>
<tr>
<td>?</td>
<td>Wildcard indicating a single character (even when enclosed in quotation marks); don’t care state; indicates a free (unforced) state with PATTERN or DEPOSIT.</td>
</tr>
<tr>
<td>@</td>
<td>Abbreviation for SET INDIRECT.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Word subscripts; macro formal parameters; DEPOSIT waveforms; encloses SELECT trigger expressions; encloses model parameters.</td>
</tr>
</tbody>
</table>
Table B–2 (Cont.)  Printing Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Up arrow: A control character specifier in text strings. However, two</td>
</tr>
<tr>
<td></td>
<td>consecutive up arrows cause a single up arrow to be displayed.</td>
</tr>
<tr>
<td>_</td>
<td>Underline: A valid character for signal names and parameter names.</td>
</tr>
<tr>
<td>\</td>
<td>Separates a module from a line number in WATCH/STATEMENT.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>{}</td>
<td>Behavior language directive specifier; command language fault specifier.</td>
</tr>
</tbody>
</table>
DECSIM Symbols and Their Character Sets

This appendix lists all DECSIM symbols and the character set for each. Unless specified in the table, definitions apply to all three languages: the Network Description Language (structure), the Behavior Description Language (behavior), and the Interactive Command Language.

Four rules apply:

- Except for command language macro names, you can use nonstandard characters (those that are not in the character set) by enclosing the name in quotation marks or apostrophes; use quotation marks in the Behavior Description Language, apostrophes in the Network Description Language, and either in the Interactive Command Language.

- Lowercase letters are treated as uppercase unless they are in text strings.

- If a name can be interpreted as a number, enclose it in quotation marks (for behavior), apostrophes (for structure), either (for command language).

- Do not begin names with the percentage sign (%) or the dollar sign ($); these are reserved for DECSIM.

Warning: Do not use period (.), question mark (?), or asterisk (*) when creating names. Although you can create names containing these characters, you might not be able to access those names from the command language.
# DECSIM Symbols and Their Character Sets

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Character Set</th>
<th>Restrictions</th>
<th>Limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label names</td>
<td>A-Z, a-z, 0-9, underscore (_), percentage sign (%)</td>
<td>In structure, do not enclose the period in apostrophes if it indicates hierarchy.</td>
<td>63 characters</td>
</tr>
<tr>
<td></td>
<td>To include nonstandard characters, do one of the following: Behavior—Enclose the name in quotation marks. Structure—Enclose the name in apostrophes. Command language—Enclose the name in quotation marks.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal names</td>
<td>A-Z, a-z, 0-9, dash (-), parentheses (), underscore (_), percentage sign (%)</td>
<td>For structure, polarity (using the dash or parentheses) can be indicated without quoting the name. Multiple consecutive spaces are treated as one space.</td>
<td>63 characters. The polarity indicator is retained when truncating a long name.</td>
</tr>
<tr>
<td></td>
<td>To include nonstandard characters, enclose the name in quotation marks. Subscripts can appear after a signal name, but are not considered part of the name. Structure—Space is a standard character. Period (.) indicates hierarchy, unless enclosed in quotation marks. Do not use periods when creating names because in the Command Language the period indicates hierarchy. Question mark (?), and asterisk (<em>) can be used by enclosing the name in quotation marks. Dash (-) and parentheses () indicate polarity. Command language—Space is not a standard character, and must be enclosed in quotation marks. Period (.) always indicates hierarchy, question mark (?) and asterisk (</em>) are always wildcards, whether they are enclosed in quotation marks or not.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter names</td>
<td>A-Z, a-z, 0-9, underscore (_), percentage sign (%)</td>
<td>Must begin with a letter unless enclosed in apostrophes.</td>
<td>135 characters</td>
</tr>
</tbody>
</table>
Table C-1 (Cont.) DECSIM Symbols and their Character Sets

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Character Set</th>
<th>Restrictions</th>
<th>Limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text strings</td>
<td>A–Z, a–z, 0–9, period (.), plus sign (+), dash (–), parentheses (), question mark (?), underscore (_), apostrophe ('), quotation mark (&quot;), pound sign (#), dollar sign ($), percentage sign (%), up arrow (^), subscripts fix and [], space, tab</td>
<td>Letters are not case–converted; text strings are displayed as they were typed. However, an up arrow (^) inserts a control character; two consecutive up arrows display a single up arrow. Apostrophes enclose all text strings. Two consecutive apostrophes cause a single apostrophe to be displayed. In all three languages, strings can simply be continued from one line to another.</td>
<td>200 characters in the command language, 135 in structure</td>
</tr>
<tr>
<td>Integer values</td>
<td>A–F, a–f, 0–9, pound sign (#), H, L, U, Z</td>
<td>The pound sign (#) is the radix indicator. When followed by #2, the characters H, L, U, and Z mean 1, 0, undefined, and high impedance.</td>
<td>H, L, U, and Z can be used only by specifying radix 2. 256 bits of binary representation. Exceptions: Absolute time = 64 bits. Relative time, qualifier arguments = 32 bits. Limited by VAX single-precision floating point format.</td>
</tr>
<tr>
<td>Parameter time values</td>
<td>0–9, pound sign (#), period (.)</td>
<td>The pound sign (#) is the radix indicator.</td>
<td>You cannot specify both a pound sign (#) and a period (.) in a time value.</td>
</tr>
<tr>
<td>Logic state values</td>
<td>Z,U,H,L, z,u,h,l, 0,1, question mark (?), pound sign (#), 2–9, A–F</td>
<td>The question mark is a wildcard character in the command language, where it can be used in SELECT trigger expressions and as an EQO operand. Used with DEPOSIT or PATTERN, the question mark can mean &quot;true.&quot; The pound sign (#) is the radix indicator.</td>
<td>Expressions in the command language are limited to 32-bit operations. 256 bits</td>
</tr>
<tr>
<td>Command language STATE name</td>
<td>A–Z, a–z, 0–9, underscore (_)</td>
<td>255 characters</td>
<td>255 characters</td>
</tr>
<tr>
<td>Command language MACRO name</td>
<td>A–Z, a–z, 0–9</td>
<td>The name must begin with a letter. Nonstandard characters are not valid.</td>
<td>200 characters</td>
</tr>
</tbody>
</table>
## Table C-1 (Cont.)  DECSIM Symbols and their Character Sets

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Character Set</th>
<th>Restrictions</th>
<th>Limits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model name for structure or</strong></td>
<td><strong>A–Z, a–z, 0–9, underscore (_), percentage sign (%)</strong></td>
<td>To include nonstandard characters, do one of the following:</td>
<td>127 characters</td>
</tr>
<tr>
<td><strong>behavior</strong></td>
<td>Behavior – Enclose the name in quotation marks.</td>
<td>Structure—Enclose the name in apostrophes.</td>
<td></td>
</tr>
<tr>
<td><strong>Subblock name for structure or</strong></td>
<td><strong>A–Z, a–z, 0–9, underscore (_), percentage sign (%)</strong></td>
<td>To include nonstandard characters, do one of the following:</td>
<td>Total of model name PLUS</td>
</tr>
<tr>
<td><strong>behavior</strong></td>
<td>Behavior—Enclose the name in quotation marks.</td>
<td>Structure – Enclose the name in apostrophes.</td>
<td>subblock name must not exceed 126 characters</td>
</tr>
<tr>
<td><strong>Behavior routine, state, synonym, macro, and label names</strong></td>
<td><strong>A–F, a–f, 0–9, underscore (_), percentage sign (%)</strong></td>
<td>You can indicate polarity, but the compiler does not recognize it.</td>
<td>63 characters</td>
</tr>
<tr>
<td><strong>Behavior module name</strong></td>
<td><strong>A–F, a–f, 0–9, underscore (_)</strong></td>
<td>The first character must be a letter. The module name must be the same as the file name.</td>
<td>21 characters is suggested.</td>
</tr>
<tr>
<td><strong>Other user-defined names</strong></td>
<td><strong>A–Z, a–z, 0–9</strong></td>
<td></td>
<td>63 characters</td>
</tr>
<tr>
<td><strong>System keywords</strong></td>
<td><strong>A–Z, a–z, 0–9, underscore (_), percentage sign (%)</strong></td>
<td>Must begin with a percentage sign. Not user-defined.</td>
<td>200 characters</td>
</tr>
<tr>
<td><strong>System variables</strong></td>
<td><strong>A–Z, a–z, 0–9, underscore (_), dollar sign ($), percentage sign (%)</strong></td>
<td>Must begin with a percentage sign. Not user-defined.</td>
<td>200 characters</td>
</tr>
<tr>
<td><strong>Transistor ID strings</strong></td>
<td><strong>A–Z, a–z, 0–9, underscore (_), percentage sign (%)</strong></td>
<td>Must be enclosed in apostrophes</td>
<td>11 characters</td>
</tr>
<tr>
<td><strong>(in PRC file)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C-4
Expression Operators

This appendix contains a thorough description of each operator.
The sections that precede the operator descriptions serve the following purposes:

- List all the operators according to relative precedence.
- Provide a table of directives and modifiers.
- Explain the WIDTH directive.
- Describe operator restrictions.
- Explain wildcard handling.
- Explain the organization of each description.
## Operators by Relative Precedence

Table D-1 lists the DECSIM expression operators in order of relative precedence and gives the functions they perform.

Precedence is indicated by an integer number. Operators with higher precedence are evaluated before operators with lower precedence. Equal precedence binary operators are evaluated left to right. Equal precedence unary operators are evaluated right to left.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Relative Precedence</th>
<th>Function</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>90</td>
<td>Multiplication</td>
<td>C, B</td>
</tr>
<tr>
<td>/</td>
<td>90</td>
<td>Division</td>
<td>C, B</td>
</tr>
<tr>
<td>+</td>
<td>80</td>
<td>Addition</td>
<td>C, B</td>
</tr>
<tr>
<td>-</td>
<td>80</td>
<td>Subtraction</td>
<td>C, B</td>
</tr>
<tr>
<td>EQL(QUOTE)</td>
<td>70</td>
<td>Equal Quote</td>
<td>C, B</td>
</tr>
<tr>
<td>NEQ(QUOTE)</td>
<td>70</td>
<td>Not Equal Quote</td>
<td>C, B</td>
</tr>
<tr>
<td>EQL, NEQ, LSS, LEQ, GTR, GEQ</td>
<td>70</td>
<td>Relational operators</td>
<td>C, B</td>
</tr>
<tr>
<td>BUF</td>
<td>60</td>
<td>Boolean operator (unary)</td>
<td>N</td>
</tr>
<tr>
<td>NOT</td>
<td>60</td>
<td>Boolean operator (unary)</td>
<td>C, B, N</td>
</tr>
<tr>
<td>AND, NAND</td>
<td>50</td>
<td>Boolean operators (binary)</td>
<td>C, B, N</td>
</tr>
<tr>
<td>OR, NOR</td>
<td>40</td>
<td>Boolean operators (binary)</td>
<td>C, B, N</td>
</tr>
<tr>
<td>EQV, XOR</td>
<td>30</td>
<td>Boolean operators (binary)</td>
<td>C, B, N</td>
</tr>
<tr>
<td>CONV4</td>
<td>20</td>
<td>Binary conversion</td>
<td>C</td>
</tr>
</tbody>
</table>

### Key
- **C**—command language
- **B**—behavior language
- **N**—network equation statements
Table D–1 (Cont.) Expression Operators by Relative Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Relative Precedence</th>
<th>Function</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONV0Z, CONV1Z,</td>
<td>20</td>
<td>Unary high-impedence</td>
<td>C, B, N</td>
</tr>
<tr>
<td>CONVZ0, CONVZ1</td>
<td></td>
<td>conversions</td>
<td></td>
</tr>
<tr>
<td>CONVU0, CONVU1,</td>
<td>20</td>
<td>Unary undefined</td>
<td>C, B, N</td>
</tr>
<tr>
<td>CONV1U, CONV0U,</td>
<td></td>
<td>conversions</td>
<td></td>
</tr>
<tr>
<td>CONVUZ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAP</td>
<td>20</td>
<td>Mapping operator</td>
<td>C, B</td>
</tr>
<tr>
<td>OXT, ZXT</td>
<td>20</td>
<td>Extension operators</td>
<td>C, B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(unary)</td>
<td></td>
</tr>
<tr>
<td>SXT</td>
<td>20</td>
<td>Extension operator</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(unary)</td>
<td></td>
</tr>
<tr>
<td>TS, TSCON, MOSTS</td>
<td>20</td>
<td>Tristate operators</td>
<td>C, B, N</td>
</tr>
</tbody>
</table>

Key

C—command language
B—behavior language
N—network equation statements

D.2 Features: Twostate and Fourstate Data Types and Directives

Most expression operators apply to both twostate and fourstate data types and accept certain Behavior Language directives. See Section 2.5 for a description of the TWOSTATE and FOURSTATE modifiers. Table D–2 shows which modifiers and directives apply to each operator.
Table D-2  Directives

<table>
<thead>
<tr>
<th>Operator</th>
<th>TWOSTATE</th>
<th>FOURSTATE</th>
<th>US</th>
<th>TC</th>
<th>WIDTH</th>
<th>%CARRY</th>
<th>%CARRYIN</th>
<th>%OVERFLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>X</td>
<td>X</td>
<td>D</td>
<td>N/A</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>- (unary)</td>
<td>X</td>
<td>X</td>
<td>N/A</td>
<td>D</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>+ (binary)</td>
<td>X</td>
<td>X</td>
<td>D</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- (binary)</td>
<td>X</td>
<td>X</td>
<td>N/A</td>
<td>D</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>BUF</td>
<td>X</td>
<td>X</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONVxx</td>
<td>N/A</td>
<td>X</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CONV4</td>
<td>X</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>EQL(QUOTE),</td>
<td>X</td>
<td>X</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEQ(QUOTE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EQL, NEQ, LSS,</td>
<td>X</td>
<td>X</td>
<td>D</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEQ, GTR, GEQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAP</td>
<td>N/A</td>
<td>X</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MOD, *, /</td>
<td>X</td>
<td>X</td>
<td>D</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td>X</td>
<td>X</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>AND, NAND, OR,</td>
<td>X</td>
<td>X</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NOR, EQV, XOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shlts</td>
<td>X</td>
<td>X</td>
<td>D</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SXT</td>
<td>X</td>
<td>X</td>
<td>D</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OXT, ZXT</td>
<td>X</td>
<td>X</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TS, TCON, MOSTS</td>
<td>N/A</td>
<td>X</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Key

D—the default directive
X—applicable
N/A—not applicable

D.3

WIDTH DIRECTIVE

The WIDTH directive overrides the default width of the result for any operation and follows the operator in the expression.

Format

```
operator(width = integer)
```

The result is truncated to the width specified in the directive if the specified width is less than the default; the least significant bits are truncated.

The result is zero-extended if the width specified in the directive is greater than the default width of the result for the given operation.
Wildcard Handling in Command Language Expressions

The wildcard (?) character is allowed in some operator expressions that use logic value constants as operands.

A wildcard (?) can represent either a zero (0) or a one (1) in an operand. Operators that allow wildcards perform logic functions as opposed to arithmetic functions.

The table below shows which expressions accept the wildcard (?) as input. If wildcards are not explicitly implemented for an operator, the ? operand bit is considered to be a U.

The wildcard (?) is available only in the Command Language.

<table>
<thead>
<tr>
<th>Operator Class</th>
<th>Wildcard Applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>+/- (unary, binary)</td>
<td>no</td>
</tr>
<tr>
<td>Boolean bit-wise (unary, binary)</td>
<td>no</td>
</tr>
<tr>
<td>Concatenation</td>
<td>yes</td>
</tr>
<tr>
<td>Conversion</td>
<td>no</td>
</tr>
<tr>
<td>Extension</td>
<td>no</td>
</tr>
<tr>
<td>MAP</td>
<td>no</td>
</tr>
<tr>
<td>Multiplication, etc.</td>
<td>no</td>
</tr>
<tr>
<td>Parentheses</td>
<td>yes</td>
</tr>
<tr>
<td>Relational</td>
<td>yes</td>
</tr>
<tr>
<td>Shifts</td>
<td>yes</td>
</tr>
<tr>
<td>Subscripting of expressions</td>
<td>no</td>
</tr>
<tr>
<td>Tristate</td>
<td>no</td>
</tr>
</tbody>
</table>

Rules

General rules are:

- Two's complement (TC) arithmetic is performed in the Command Language by default. Command language expressions are accurate to 32 bits or less. TC arithmetic has been implemented for twostate signals of 32 bits or less in behavior only.

- Unsigned (US) arithmetic is performed in the Behavior Language by default; exceptions are noted in the specific operator descriptions.

- Operator directives are applicable only to the Behavior Language.

- Wildcards are applicable only to the Command Language.

- Network Equation expressions are limited to a width of one bit.
• Command Language expressions are limited to 32 bits. Behavior Language expressions are limited to 256 bits.

### D.6 Organization

The operator descriptions are grouped according to their function. Unary and binary operators are included under the same heading in separate subsections.

The descriptions consist of these components:
• One-line definition of the operator's function
• Listing of the languages to which an operator applies
• Expanded description of the operator
• Listing of the operator directives (for Behavior Language)
• The default width of the result
• Truth table
• Explanation of how undefined bits are handled
• Suggestions to explain how to avoid common mistakes
• Examples

### D.7 Assignment

The assignment operator (=) passes the value of an expression to a variable.

**Application**

Command Language, Behavior Language, Network Equation statements

**Format**

```
variable = [{directive}] expression
```

**Description**

This operator passes the value of an expression on the right to a variable on the left. Twostate values may be assigned to fourstate values. Fourstate values may be assigned to twostate values. Mixed fourstate and twostate values may also be assigned.
Expression Operators

Example

\[ b_2 = a_4 + b_4 \] \hspace{1cm} !fourstate assigned to twostate. \( U\#2 \) and \( Z\#2 \) change to 0 by default

\[ c_2 = \text{CONVZ1} \ c_4 \] \hspace{1cm} !2 to 1, \( U \) to 0 by default

\[ d_2 = (\text{CONVZ1} \ c_4) + b_2 \] \hspace{1cm} !value of fourstate \( c_4 \) plus twostate \( b_2 \) assigned to twostate \( d_2 \)

Behavior Language Directives

Assignment directives act only on the local assignment, not on the expression to the right of the directive.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>(time)</td>
<td>Specifies delays. ( \text{&quot;Time&quot;} ) can be a floating-point number, a delay parameter, or an expression. The default unit is a nanosecond. Use the ( \text{(TIMESCALE)} ) directive with ( \text{(time)} ) to change the default units. See Section 2.8.2 for details.</td>
</tr>
<tr>
<td>(TC)</td>
<td>Specifies two’s complement arithmetic.</td>
</tr>
<tr>
<td>(US)</td>
<td>Specifies unsigned arithmetic.</td>
</tr>
</tbody>
</table>

Examples

\[ \text{out\_port} = (10) \ a \ + \ b; \]

\[ \text{global\_state} = (5 \ \ast \ \text{cycle}) \ c \ \& \ \& \ d; \]

Width of Result

In an assignment, if the width of the expression result is less than the width of the assignment target variable, the width of the expression result is extended with zeros. However, if the width of the expression result is greater than the width of the target variable, the most significant bits of the result are truncated.

Examples

1. \text{SIM> STATE A<0:15>} \hspace{1cm} !Command Language
   \text{SIM> STATE B<0:15>}
   \text{SIM> STATE C<0:15>}
   \text{SIM> EVAL C = 111\#2 + 11111\#2}
   \text{SIM> EVAL C = 38}

2. \text{SIM> RADIX 2}
   \text{SIM> EVAL C = 111\#2 + 11111\#2}
   \text{SIM> EVAL C = 100110}
Expression Operators

3. SIM> EVAL A = 101#2
SIM> EVAL B = 01010#2
SIM> EVAL C = A + B
SIM> EVAL C
00000000000UUU1

4. MODEL test = ;
   BEHAVIOR;
   STATE x<7:0>,
       y<7:0>,
       z<7:0>,
       e<7:0>;
   ROUTINE addition( );
   x = 11111111;
   y = 1;
   z = (20) x + (e = %CARRY) y;
   !Assignment of an expression
   !to z and %CARRY to e.
   ENDRoutine addition;
   ENDBehavior;
   ENDMODEL test;

SIM> COMBEH test
SIM> COMPILE test
SIM> SCOPE all
SIM> CALL addition
SIM> SYMBOLS
   E<7:0> = 11111111
   X<7:0> = 11111111
   Y<7:0> = 1
   Z<7:0> = 0
   ! Most significant bit of
   ! 9-bit result is truncated

D.8 Arithmetic Operators

The arithmetic operators are grouped in the following categories:
- Addition and subtraction
- Multiplication, division, and MODULO

D.8.1 Addition and Subtraction Operators

The unary minus(−) and binary plus (+) and minus (−) operators are described in the following sections.

D.8.1.1 Unary Minus

Unary minus calculates the arithmetic negation of the operand.

Application

Command Language, Behavior Language
Expression Operators

Format
- [ {directive} ] operand

Description
The unary minus finds the arithmetic negation of the operand. Two's complement arithmetic is used to determine the result.

Note: In the Command Language, you must subtract a variable from zero in order to get its arithmetic negation. See Examples 3 and 4.

Behavior Language Directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>(TC)</td>
<td>Performs logical NOT and adds 1. %OVERFLOW can be set; %CARRY cannot. (TC) is the default.</td>
</tr>
<tr>
<td>(variable=%OVERFLOW)</td>
<td>Stores the overflow bits resulting from the operation. Used with (TC).</td>
</tr>
<tr>
<td>(WIDTH=integer)</td>
<td>Specifies width of expression. See Section D.3 for details.</td>
</tr>
</tbody>
</table>

Note: Constants with a preceding minus sign interpret the minus sign as a part of the constant, not as an operator.

Unary minus for TC sets %OVERFLOW if the operand is the largest negative integer that has no positive counterpart. In this case, the result is unchanged.

Width of Result
Width of the operand plus 1.

Truth Table for a single–bit operand

<table>
<thead>
<tr>
<th>operand</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0#2</td>
<td>00#2</td>
</tr>
<tr>
<td>1#2</td>
<td>11#2</td>
</tr>
<tr>
<td>Z#2</td>
<td>UU#2</td>
</tr>
<tr>
<td>U#2</td>
<td>UU#2</td>
</tr>
</tbody>
</table>
Expression Operators

Undefined Handling
A TC undefined bit operand yields an undefined bit result.

Examples
In the following examples, a and b are defined to have 8 bits: a<7:0>, b<7:0>.

1. SIM> RADIX 2
   SIM> EVAL -10101010#2
       10101010

2. SIM> EVAL -1010101#2
       1010101

3. SIM> STATE a<7:0>
   SIM> EVAL a = 1010101#2
   SIM> EVAL 0 - a
       11111111111111111111111101010111

4. SIM> STATE b<7:0>
   SIM> EVAL b = 0 - a
   SIM> EVAL b
       10101011

5. a = -b
   where b = 00000001#2
   Result: a = 11111111#2 ! Width of b is truncated to
   ! 8 bits when assigned to a.

6. a = -b
   where b = 1111111#2
   Result: a = 100000001#2

7. a = -(WIDTH = 2) b
   where b = 00000001#2
   Result: a = 11#2

D.8.1.2 Binary Addition and Subtraction
The binary plus (+) and minus (−) perform addition and subtraction on two
operands.

Application
Command Language, Behavior Language

Format
operand1 + [ {directive} ] operand2
operand1 − [ {directive} ] operand2
Expression Operators

Description

Sign bit interpretations for a constant should be supplied by an explicit minus sign.

Unsigned (US) subtraction is treated as two's complement (TC) subtraction.

Behavior Language Directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>(TC)</td>
<td>Specifies two’s complement arithmetic. TC subtraction sets the %CARRY flag to give the bits borrowed.</td>
</tr>
<tr>
<td>(US)</td>
<td>Specifies unsigned arithmetic. For addition only.</td>
</tr>
<tr>
<td>{variable=%CARRY}</td>
<td>Captures the carry-out bits.</td>
</tr>
<tr>
<td>{%CARRY</td>
<td>=variable}</td>
</tr>
<tr>
<td>{variable=%OVERFLOW}</td>
<td>See Section 2.8.4.4 of this manual.</td>
</tr>
<tr>
<td>{WIDTH=integer}</td>
<td>Specifies the width of the expression. See Section D.3.</td>
</tr>
</tbody>
</table>

Width of Result

Behavior Language – width of the wider operand.
Command Language – width of the wider operand + 1.

Truth Table

<table>
<thead>
<tr>
<th>operand1 + operand2</th>
<th>Single-bit Result</th>
<th>Behavior %CARRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>Z</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>U</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>U</td>
</tr>
<tr>
<td>Z</td>
<td>1</td>
<td>U</td>
</tr>
<tr>
<td>Z</td>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>Z</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>
Expression Operators

<table>
<thead>
<tr>
<th>operand1 + operand2</th>
<th>Single-bit Result</th>
<th>Behavior %CARRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

Undefined Handling

If there are U's or Z's in either operand, two answers are calculated, the highest possible answer (where U's and Z's in the operands are converted to 1's) and the lowest possible answer (where U's and Z's are converted to 0's). U's are placed in the range of bits with inconsistent results.

Examples

1. SIM> RADIX 2
   SIM> EVAL 100001#2 + 1010#2
       1UU011#2
   !Command Language
   !with undefined states

2. SIM> RADIX 2
   SIM> EVAL 100001#2 - 1010#2
       10111#2

3. SIM> EVAL 1110001#2 - 0001111#2
       1100010

4. SIM> EVAL UUU010111#2 + 01010101#2
       UUU000000

In the Behavior Language examples, the variables are 8 bits wide: x<7:0>, y<7:0>, z<7:0>, b<7:0>.

5. z = x + (TC, b = %OVERFLOW) y
   where x = 101011#2 and y = 101011#2
   Result: z = 1010110#2 and b = 1111101#2

6. z = x + (c = %CARRY) y
   where x = 101011#2 and y = 101011#2
   Result: z = 1010110#2 and c = 101011#2

7. z = x -(c = %CARRY) y
   where x = 100000#2 and y = 111#2
   Result: z = 11001#2 and c = 11111#2

8. z = x - y !Specify FOURSTATE
   where x = 1111UUU#2 - 1111#2 !modifier
   Result: z = 1UUUUUU#2

D.8.2 Multiplication, Division, and Modulo Operators

The multiplication (*), division (/), and modulo (MOD) operators perform their arithmetic functions on two operands.
Expression Operators

Application
Command Language, Behavior Language

Format
operand1 * [ {directive} ] operand2
operand1 / [ {directive} ] operand2
operand1 MOD operand2

Behavior Language Directives
The directive [WIDTH=integer] specifies the width of the expression. See Section D.3 for more information.

Width of Result
The table below shows the resulting width for each operation.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Width of Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>width operand1 + width operand2</td>
</tr>
<tr>
<td>/</td>
<td>width of operand1</td>
</tr>
<tr>
<td>MOD</td>
<td>width of operand2</td>
</tr>
</tbody>
</table>

Truth Table for One-Bit Operations

<table>
<thead>
<tr>
<th>Operand1</th>
<th>Operand2</th>
<th>*</th>
<th>/</th>
<th>MOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>Z</td>
<td>1</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>0</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>U</td>
<td>1</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>0</td>
<td>0</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>?</td>
<td>1</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>?</td>
<td>0</td>
<td>0</td>
<td>error</td>
<td>error</td>
</tr>
</tbody>
</table>
Expression Operators

Undefined Handling

For all operations, U's and Z's in the second operand result in U's for the entire width of the result.

For multiplication, U's and Z's in either operand result in U's for the entire width of the result.

For division and MOD, if there are U's or Z's in the first operand, DECSIM substitutes U's for the bit values from the most significant bit that contains a U to the least significant bit.

Errors

In the Behavior Language, division by zero causes an arithmetic trap, which returns a value of zero and exits you from the model.

Examples

1. SIM> RADIX 2
   SIM> EVAL 10001#2 * 1001#2
        10011001
   !Command Language

2. SIM> EVAL 1001#2 / 11#2
        11

3. SIM> EVAL 1001#2 MOD 10#2
        1

4. SIM> EVAL 1U1#2 / 10#2
        0UU

5. SIM> RADIX 10
   SIM> EVAL 10 / 4
        2

6. SIM> RADIX 2
   SIM> EVAL 11U0#2/10#2
        11U#2
   !Undefined bit in division

7. SIM> RADIX 2
   SIM> EVAL 1001#2/10#2
        UUU#2
   !Undefined bits

D.9 Bit Subscript Operator

In statements or expressions, the bit subscript operator specifies a particular subset of a state, port, constant, or subexpression that you want to access.

Application

Command Language, Behavior Language, Network Equation expressions
Expression Operators

Format

state <subscript>
port <subscript>
(expression)<subscript>
(constant)<subscript>

Where subscript is a single bit, a range of bits, or a list of bits or ranges of bits.

Description

You can extract the individual bit values assigned to a particular variable by indicating the bits to be accessed. For example, a<15> checks the most significant bit of a<15:0>; a<3:0> checks the four least significant bits. A<3:0> has the same effect as concatenating A<3> and A<0>.

A constant or the result of an expression has implicit bit subscripts. To extract a subset of an expression or a constant, enclose the expression or constant in parentheses. See Section 2.7.9.

To obtain consistent results when extracting bits from state and ports, use the same bit order that was used in the state or port declaration. For example, if a state is declared A<15:0>, you should access subsets of that state in high-to-low order (A<3:0>, not A<0:3>).

Note: Do not confuse bit subscripts with word subscripts. Word subscripts are used only with arrays.

Behavior Language Directives

None

Width of Result

Total number of bits specified or extracted.

Undefined and Wildcard Handling

The value of the expression is transmitted unchanged for U, Z, and ? bits. However, an error is produced when the subscript index is U, Z, or ?.

Examples

1. SIM> STATE A<0:3>

   !Command Language

2. SIM> STATE A<15:0>
   SIM> RADIX 2
   SIM> EVAL A = 11111111#2
   SIM> EVAL A
       11111111
   SIM> EVAL A<0:15>

   !Reverses the order of the bits.

       1111111110000000
Expression Operators

3. SIM> RADIX 2
   SIM> STATE A<15:0>
   SIM> EVAL A = 11100011110001110#2
   SIM> EVAL A<12> !Checks value of specified bit.
      0

4. SIM> RADIX 2
   SIM> STATE A<15:0>
   SIM> EVAL A =1010101011111#2
   SIM> EVAL A<15:9> !Prints the most significant bits of A.
      1010 !Note that the three 0’s to the left
      10101 !of 1010 are not printed
   SIM> EVAL A<15:8>
      101111
   SIM> EVAL A<6:0>
      101111 !Prints the least significant bits.

D.10  Boolean Operators

There are two types of Boolean operators: unary and binary. The Boolean operators act on each bit of an operand.

D.10.1 Unary Boolean Operators (Single-Bit Result)

The BUF operator provides the value of the operand if that value is an integer, otherwise it provides an undefined value.

Application

Network equation expressions

Format

BUF operand

Description

DECSIM provides the BUF operator for network equation expressions so that buffers can be modeled with timing parameters. The BUF operator's result is a 1 or a 0 if the operand is 1 or 0. If the operand has any other value the result is U.

Truth Table

<table>
<thead>
<tr>
<th>Operand</th>
<th>BUF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>?</td>
<td>U</td>
</tr>
</tbody>
</table>
D.10.2 Unary Boolean Operators (Multiple–Bit Result)

The NOT operator performs a Boolean operation, one bit at a time, on all bits of the specified operand; consequently, it can produce multiple–bit results.

Application

Command Language, Behavior Language, Network Equation expressions

Format

NOT operand

Description

DECSIM provides the unary Boolean operator NOT. The NOT operator gives the one's complement of the operand, thus simulating the function of an inverter.

Behavior Language Directives

None.

Width of Result

Width of the operand.

Truth Table

The truth table on a per bit basis is as follows:
Expression Operators

<table>
<thead>
<tr>
<th>Operand</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>?</td>
<td>U</td>
</tr>
</tbody>
</table>

Undefined Handling

If the operand is undefined (U), the result is undefined. If the operand is a Z, the result is undefined.

Suggestions

Because the precedence of the NOT operator is lower than the precedence of the arithmetic and relational operators, specify which operations are to be executed first with parentheses (). For example: (NOT A) + B.

Examples

1. `SIM> RADIX 2
   SIM> EVAL NOT 111100000ZZZZUUUU"2`  
   `00001111UUUUUUUU`
2. `SIM> EVAL NOT 1111UUU????"2`  
   `0000UUUUUU`
3. `SIM> EVAL NOT 111"2`  
   `0`
4. `MODEL fun out = A;`  
   `EQN out = NOT A;`  
   `ENDMODEL fun;`  
   `where A = 1`  
   `Result: out = 0`

D.10.3 Binary Boolean Operators (Multiple–Bit Result)

These operators perform bit–wise Boolean operations on two operands. A bit–wise operation checks all corresponding bits, bit by bit, contained within the two operands.

Application

Command Language, Behavior Language, Network Equation expressions
Expression Operators

Format

operand1 AND operand2
operand1 NAND operand2
operand1 OR operand2
operand1 NOR operand2
operand1 XOR operand2
operand1 EQV operand2

Description

These binary Boolean operators simulate the logical functions of the following gates: AND, OR, NAND, NOR, and XOR. EQV checks if the bit values in the two operands are the same, producing the complement of the result of XOR.

Behavior Language Directives

None

Width of Result

Width of the wider operand.

Truth Table

<table>
<thead>
<tr>
<th>operand1</th>
<th>operand2</th>
<th>AND</th>
<th>NAND</th>
<th>OR</th>
<th>NOR</th>
<th>XOR</th>
<th>EQV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>Z</td>
<td>0</td>
<td>1</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>0</td>
<td>U</td>
<td>0</td>
<td>1</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>0</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Z</td>
<td>U</td>
<td>U</td>
<td>1</td>
<td>0</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>1</td>
<td>0</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
<td>U</td>
<td>U</td>
<td>1</td>
<td>0</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>
## Expression Operators

<table>
<thead>
<tr>
<th>operand1</th>
<th>operand2</th>
<th>AND</th>
<th>NAND</th>
<th>OR</th>
<th>NOR</th>
<th>XOR</th>
<th>EQV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>Z</td>
<td>1</td>
<td>U</td>
<td>U</td>
<td>1</td>
<td>0</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>Z</td>
<td>Z</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>Z</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>Z</td>
<td>?</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>1</td>
<td>U</td>
<td>U</td>
<td>1</td>
<td>0</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>Z</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>?</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>?</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>?</td>
<td>1</td>
<td>U</td>
<td>U</td>
<td>1</td>
<td>0</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>?</td>
<td>Z</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>?</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

**Note:** Wildcards (?) are treated as U operands.

### Undefined Handling

An undefined (U) bit or a Z bit in an operand produce undefined results, except for these cases:

- U in one operand ANDed with 0 in the other operand produces 0.
- U in one operand NANDed with 0 in the other operand produces 1.
- U in one operand ORed with 1 in the other operand produces 1.
- U in one operand NORed with 1 in the other operand produces 0.

### Suggestions

Do not use the equal (=) sign in place of EQV; the equal sign is used for assigning values.

### Examples
D.11 **Concatenation Operator**

The concatenation operator joins two operands together, placing the second operand to the right of the first.

**Application**

Command Language, Behavior Language

**Format**

```
operand1 & operand2
```

**Description**

Bit fields are concatenated regardless of sign bit interpretations. Specifically, the sign bit of an operand on the right of a concatenation operator loses its functional significance.

You can concatenate bits or words extracted from a variable or an array. However, the Network Language allows only bit concatenation.

For the Behavior Language, widths of operands must be constant. For example, the expression `X<1:J> & Y<K:L>` is illegal. See "Suggestions" below for a workaround.

The value of `(A<31:0> & B<31:0>) SR0 32 is always 0, because the shift operation has an operand of only 32 bits.

**Note:** In the Interconnect Language, the ampersand (`&`) symbol is used to specify the joining of signal names, but it is not an operator. See Section 3.4.6 for more information.
Behavior Language Directives

(WIDTH=integer) Specifies width of expression.
See Section D.3 for details.

Width of Result
Sum of the widths of the two operands.

Truth Table

<table>
<thead>
<tr>
<th>Operand &amp; Operand</th>
<th>Result (2 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>00</td>
</tr>
<tr>
<td>0 1</td>
<td>01</td>
</tr>
<tr>
<td>0 Z</td>
<td>0Z</td>
</tr>
<tr>
<td>0 U</td>
<td>0U</td>
</tr>
<tr>
<td>0 ?</td>
<td>0?</td>
</tr>
<tr>
<td>1 0</td>
<td>10</td>
</tr>
<tr>
<td>1 1</td>
<td>11</td>
</tr>
<tr>
<td>1 Z</td>
<td>1Z</td>
</tr>
<tr>
<td>1 U</td>
<td>1U</td>
</tr>
<tr>
<td>1 ?</td>
<td>1?</td>
</tr>
<tr>
<td>Z 0</td>
<td>Z0</td>
</tr>
<tr>
<td>Z 1</td>
<td>Z1</td>
</tr>
<tr>
<td>Z Z</td>
<td>ZZ</td>
</tr>
<tr>
<td>Z U</td>
<td>ZU</td>
</tr>
<tr>
<td>Z ?</td>
<td>Z?</td>
</tr>
<tr>
<td>U 0</td>
<td>U0</td>
</tr>
<tr>
<td>U 1</td>
<td>U1</td>
</tr>
<tr>
<td>U Z</td>
<td>UZ</td>
</tr>
<tr>
<td>U U</td>
<td>UU</td>
</tr>
<tr>
<td>U ?</td>
<td>U?</td>
</tr>
<tr>
<td>? 0</td>
<td>?0</td>
</tr>
<tr>
<td>? 1</td>
<td>?1</td>
</tr>
<tr>
<td>? Z</td>
<td>?Z</td>
</tr>
<tr>
<td>? U</td>
<td>?U</td>
</tr>
<tr>
<td>? ?</td>
<td>??</td>
</tr>
</tbody>
</table>

Note: "?" is recognized only in the Command Language.
Expression Operators

Undefined Handling

All four states are unchanged when concatenated.

Suggestions

You cannot concatenate two variables with inconsistent subscripts; however, you can join the two variables together in two assignment statements by calculating the placement of each with bit subscripts.

Examples

1. \[ z_{a:b} = x_{i:j} \& y_{k:m} \]
   \[ z_{(i+j+1)+(k+m+1)-1:(k+m+1)} = x_{i:j} \]
   \[ z_{(k-m):0} = y_{k:m} \]
   !Will not concatenate.
   !Places x in the most significant bits of z.
   !Places y in the least significant bits of z.

The following is an example with constants:

2. SIM> STATE x<7:0>
   SIM> STATE y<15:0>
   SIM> STATE z<23:0>
   SIM> RADIX 2
   SIM> EVAL x = 11110102
   SIM> EVAL z<23:16> = x
   SIM> EVAL y = 1012
   SIM> EVAL z<15:0> = y
   SIM> EVAL z
   111101000000000000000001
   !Because 1012 is only 3 bits wide
   !and y is 16 bits wide, DECSIM
   !fills in the other 13 with zeros.

3. SIM> RADIX 2
   SIM> EVAL F016 & 02
   111100000U

4. SIM> STATE a<15:0>
   SIM> STATE array[0:1]<7:0>
   SIM> EVAL array[0]=000000002
   SIM> EVAL array[1]=111111112
   SIM> EVAL a = array[1]&array[0]
   SIM> SHOW STATE
   A<15:0> = 1111111100000000
   ARRAY[0:1]<7:0>

5. a & b
   where a = 00012
   b = 00010012
   Result: 000100010012

6. a & {width = 8} b
   where a = 00012
   b = 0101002
   Result: 010100002

D-23
D.12 Conversion Operators

There are two kinds of conversion operators, unary and binary. There are two kinds of unary conversion operators, high-impedance and undefined.

Format Summary

operand CONV4 operand
CONV02 operand
CONV1Z operand
CONVZ0 operand
CONVZ1 operand
CONVU0 operand
CONVU1 operand
CONV1U operand
CONV0U operand
CONVU2 operand

D.12.1 Binary Conversions

The CONV4 binary conversion operator converts TWOSTATE to FOURSTATE.

Application

Command Language

Description

The CONV4 operator takes two operands whose values must be 0 or 1 and produces a result of 0, 1, U, or Z. It does not produce a value of ? (wildcard).

Truth Table

<table>
<thead>
<tr>
<th>Operand 1</th>
<th>CONV4</th>
<th>Operand 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>U</td>
</tr>
</tbody>
</table>

Undefined Handling

The operands values must be 0 or 1.
D.12.2 High-Impedance and Undefined Conversions

The high-impedance and undefined conversion operators perform data conversions for internal coding of U and Z states.

Application

Command Language, Behavior Language, Network Equation expressions. For fourstate data types only.

Description

The conversion operators convert one bit value in the operand into another bit value in the result.

These operators are abbreviations. CONVxx is an abbreviation for convert. The last two bit values indicate the value to be converted and the value to which it is to be converted. For example, CONVUZ expands to mean: convert the U's in the operand to Z's in the result.

The conversion operators can be cascaded to formulate complex conversions. See "Suggestions" below for a method of converting Z's to U's.

Behavior Language Directives

None

Width of Result

Width of the operand.
Expression Operators

Truth Table

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operand</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CONVOZ</td>
<td>Z</td>
<td>1</td>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>CONVOU</td>
<td>U</td>
<td>1</td>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>CONV1Z</td>
<td>0</td>
<td>Z</td>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>CONV1U</td>
<td>0</td>
<td>U</td>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>CONVZ0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>U</td>
</tr>
<tr>
<td>CONVZ1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>U</td>
</tr>
<tr>
<td>CONVU0</td>
<td>0</td>
<td>1</td>
<td>Z</td>
<td>0</td>
</tr>
<tr>
<td>CONVU1</td>
<td>0</td>
<td>1</td>
<td>Z</td>
<td>1</td>
</tr>
<tr>
<td>CONVUZ</td>
<td>0</td>
<td>1</td>
<td>Z</td>
<td>Z</td>
</tr>
</tbody>
</table>

Suggestion

To convert Z's to U's, OR the operand with zero (0). For example:

SIM> EVAL 10ZU$2 OR 0000
100U$2

Examples

1. SIM> RADIX 2
   SIM> EVAL CONVOZ 0001111$2
       ZZZ1111

2. SIM> EVAL CONV1U 11110000$2
       UUUU0000

3. SIM> EVAL CONV2E 1E01E01E$2
       10010010

4. MODEL U_detect out = in;
   EQN out = CONVZ0 CONV1Z in;
   "out" is U if "in" is U,
   0 otherwise.

D.13 Extension Operators

The extension operators expand the width of the result.

Application

Command Language, Behavior Language
SXT is implemented in the Behavior Language only.
Expression Operators

Format

SXT [ {directive} ] operand
ZXT [ {directive} ] operand
OXT [ {directive} ] operand

Description

The sign extension operator (SXT) extends the result so it is numerically equivalent to the original operand, but has a different width. The one extension (OXT) and zero extension (ZXT) operators splice 1's and 0's onto the left of the operand.

In the Behavior Language, SXT must be used with the directive (TC).

Behavior Language Directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>(TC)</td>
<td>Specifies two's complement for SXT.</td>
</tr>
</tbody>
</table>

Width of Result

In the Command Language, the result is extended to 32 bits by default. The Command Language does not accept widths greater than 32 bits.

In the Behavior Language, the result is extended to 256 bits by default.

Truth Table

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZXT operand</td>
<td>0's &amp; operand</td>
</tr>
<tr>
<td>OXT operand</td>
<td>1's &amp; operand</td>
</tr>
<tr>
<td>SXT(TC) operand</td>
<td>most significant bit repeated __operand</td>
</tr>
</tbody>
</table>

Undefined Handling

With the ZXT and OXT operators, zeros or ones are added to the left. The original bits are unchanged.

With the SXT operator, a U in the sign-bit position is extended as a string of U's; U's in the magnitude portion are unchanged.
Examples

1. SIM> RADIX 2
   SIM> EVAL OXT 001#2
   1111111111111111111111111111111001

In the following two Behavior Language examples, a and b are 32 bits, c is 16 bits, and the radix is 16.

2. a = OXT b
   where b = FF#16 !STATE b<31:0>
   Result: FF#16

3. a = OXT c
   where c = FF#16 !STATE c<15:0>
   Result: FFFF00FF#16

4. z = OXT c !STATE z<255:0>
   where c = FF#16
   Result: FFFFFFFFFFFFFFFFFFFFFFFFF00FF#16

D.14 Mapping Operator

The MAP operator converts the states in the second operand according to the designation in the first operand.

Application

Command Language, Behavior Language.
For fourstate data types only.

Format

operand1 MAP operand2

Description

The MAP operator changes the states in the second operand according to the conversion information contained in the first operand's 4-bit wide map vector.

The first operand consists of four bits. In the Command Language, the most significant bit specifies the state to which the 0's in the second operand are to be converted. The next bit specifies the state for the 1's in the second operand. The third bit specifies the state for the Z's, and the least significant bit specifies the state for the U's.

For the Behavior Language, the bit order of operand1 is UZI0#2. That is, the most significant bit specifies the state to which the U's in the second operand are to be converted. The next bit specifies the state for the Z's in the second operand. The third bit specifies the state for the 1's, and the least significant bit specifies the state for the 0's.

The second operand can have any width.
Expression Operators

The MAP operator configures a custom convert operator. Some conversions are already supplied by DECSIM (see Section D.12 Conversion Operators). Wildcards (?) are not converted by MAP; instead, MAP treats them as U bits.

Behavior Language Directives
None

Width of Result
Width of the second operand.

Truth Table

<table>
<thead>
<tr>
<th>conversion</th>
<th>operand1 (map vector)</th>
<th>operand2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01ZU</td>
<td>0011ZZUU</td>
<td>0011ZZUU</td>
</tr>
<tr>
<td>CONVZU</td>
<td>01UU</td>
<td>0011ZZUU</td>
<td>0011UUUU</td>
</tr>
<tr>
<td>CONV1U</td>
<td>0UZU</td>
<td>0011ZZUU</td>
<td>00UUZZUU</td>
</tr>
<tr>
<td>CONV0U</td>
<td>U1ZU</td>
<td>0011ZZUU</td>
<td>UU11ZZUU</td>
</tr>
<tr>
<td>CONVUZ</td>
<td>01ZZ</td>
<td>0011ZZUU</td>
<td>0011ZZZ</td>
</tr>
<tr>
<td>CONV1Z</td>
<td>0ZZU</td>
<td>0011ZZUU</td>
<td>00ZZZZUU</td>
</tr>
<tr>
<td>CONV0Z</td>
<td>Z1ZU</td>
<td>0011ZZUU</td>
<td>ZZ11ZZUU</td>
</tr>
<tr>
<td>CONVU1</td>
<td>01Z1</td>
<td>0011ZZUU</td>
<td>0011ZZ11</td>
</tr>
<tr>
<td>CONVZ1</td>
<td>011U</td>
<td>0011ZZUU</td>
<td>001111UU</td>
</tr>
<tr>
<td>CONV01</td>
<td>11ZU</td>
<td>0011ZZUU</td>
<td>1111ZZUU</td>
</tr>
<tr>
<td>CONVU0</td>
<td>01Z0</td>
<td>0011ZZUU</td>
<td>0011ZZ00</td>
</tr>
<tr>
<td>CONVZ0</td>
<td>010U</td>
<td>0011ZZUU</td>
<td>001100UU</td>
</tr>
<tr>
<td>CONV10</td>
<td>00ZU</td>
<td>0011ZZUU</td>
<td>0000ZZUU</td>
</tr>
</tbody>
</table>

Note: If the first operand is not 4 bits, MAP truncates the result.

Examples

1. SIM> RADIX 2
   SIM> EVAL 0110#2 MAP 1111000000000000#2
   1111000011110000

2. SIM> EVAL UU10#2 MAP 1111000000000000#2
    UUUUUUU011110000

3. SIM> EVAL 1100#2 MAP UUZ1100#2
    1111
D.15 Parentheses Operator

The parentheses ( ) operator groups operators and operands together to modify the default execution order.

Application
Command Language, Behavior Language, Network Equation statements

Format

( subexpression )

Description
The parentheses () change the order of execution designated by the rules of precedence. (See Section D.1 for a complete listing of operators and their precedence.)

An expression that is within parentheses is executed before any other expressions in a command. If there is more than one pair of parentheses, the operations within parentheses are executed in the order in which they appear in the line from left to right. If there are nested parentheses, the operation within the innermost pair is performed first.

Parentheses convert a constant into an intermediate value (subexpression). This is important for the unary minus operation.

The notation ( subexpression )<n:0> is acceptable; however, if there are variables declared in the subexpression, the number of bits assigned to the individual variables must be consistent with the number of bits assigned to the subexpression. See Section D.9 for an explanation of the subscript operator (<>).

Behavior Language Directives
None

Width of Result
Width is unchanged.

Suggestions
Parentheses make the order of precedence explicit. Since there is no limit on the number of parentheses, it is helpful to use them wherever the order of precedence is not clear.
Examples

The first Command Language example shows an expression with parentheses; the second shows an expression with the same operators and variables but without parentheses.

1. SIM> RADIX 10
   SIM> EVAL (5*9)/10+(20-6)/3
   8
2. SIM> EVAL 5*9/10+20-6/3
   22
3. z = (x + y)/y + (x * (x+y)); !Behavior Language
   where x = 6 and y = 9
   Result: z = 91
4. z = x + y/y + x + y; !Same as 3, but without
   where x = 6 and y = 9 !parentheses
   Result: z = 52
5. MODEL funl out = A, B;
   EQN out = NOT (A AND B);
   ENDMODEL funl;
   where A = 1 and B = 0
   Result: out = 1

D.16 Relational Operators

Relational operators compare two operands and produce a one-bit result of 1, 0, or U#2.

Application

Command Language, Behavior Language

Format

operand1 EQL [ (directive) ] operand2
operand1 NEQ [ (directive) ] operand2
operand1 GTR [ (directive) ] operand2
operand1 GEQ [ (directive) ] operand2
operand1 LESS [ (directive) ] operand2
operand1 LEQ [ (directive) ] operand2
operand1 EQL [ (QUOTE) ] operand2
operand1 NEQ [ (QUOTE) ] operand2

Description

The relational operators provide the following comparisons between two numbers: equal, not equal, greater than, greater than or equal, less than, less than or equal, identical, and not identical.

EQL[QUOTE] and NEQ[QUOTE] are primarily used with fourstate values. The distinction between EQL and EQL[QUOTE] is that EQL interprets U as a 1 or a 0 whereas EQL[QUOTE] checks for identical matches between bits in the two operands, including U's and Z's. Similarly, NEQ[QUOTE]
Expression Operators

checks for bits that do not match. Both operators can be abbreviated: EQL(QUOTE) to EQLQ and NEQ(QUOTE) to NEQQ.

The binary form of the operators can be used in all expressions; the unary form can be used only in SELECT trigger expressions.

All relational operators treat Z#2 as U#2, except EQL(QUOTE) and NEQ(QUOTE), which leave Z's and U's for exact matching.

Behavior Language Directives

None

Width of Result

Width is always one bit.

Truth Table

<table>
<thead>
<tr>
<th>Operand</th>
<th>Operand</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>EQL NEQ GTR GEQ LSS LEQ EQL(QUOTE) NEQ(QUOTE)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0 0 1 0 1 1 1 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0 1 0 0 1 1 0 1</td>
</tr>
<tr>
<td>0</td>
<td>Z</td>
<td>U U 0 U U 1 0 1</td>
</tr>
<tr>
<td>0</td>
<td>U</td>
<td>U U 0 U U 1 0 1</td>
</tr>
<tr>
<td>0</td>
<td>?</td>
<td>1 1 0 1 1 1 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0 1 1 1 0 0 0 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 0 0 1 0 1 1 0</td>
</tr>
<tr>
<td>1</td>
<td>Z</td>
<td>U U U 1 0 U 0 1</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
<td>U U U 1 0 U 0 1</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
<td>1 1 1 1 0 1 0 1</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>U U U 1 0 U 0 1</td>
</tr>
<tr>
<td>Z</td>
<td>1</td>
<td>U U 0 U U 1 0 1</td>
</tr>
<tr>
<td>Z</td>
<td>Z</td>
<td>U U U U U U U 1 0</td>
</tr>
<tr>
<td>Z</td>
<td>U</td>
<td>U U U U U U U 0 1</td>
</tr>
<tr>
<td>Z</td>
<td>?</td>
<td>1 1 1 1 1 1 0 1</td>
</tr>
<tr>
<td>U</td>
<td>0</td>
<td>U U U 1 0 U 0 1</td>
</tr>
<tr>
<td>U</td>
<td>1</td>
<td>U U 0 U U 1 0 1</td>
</tr>
<tr>
<td>U</td>
<td>Z</td>
<td>U U U U U U U 0 1</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U U U U U U U 1 0</td>
</tr>
<tr>
<td>U</td>
<td>?</td>
<td>1 1 1 1 1 1 0 1</td>
</tr>
<tr>
<td>?</td>
<td>0</td>
<td>1 1 1 1 1 1 1 0</td>
</tr>
<tr>
<td>?</td>
<td>1</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>?</td>
<td>Z</td>
<td>1 1 1 1 1 1 0 1</td>
</tr>
<tr>
<td>?</td>
<td>U</td>
<td>1 1 1 1 1 1 0 1</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>1 1 1 1 1 1 1 0</td>
</tr>
</tbody>
</table>

Undefined Handling

For all Relational operators except EQL(QUOTE) and NEQ(QUOTE), the U value represents either a 1 or a 0 in an operand. If some combination of U values makes the relation true when a different substitution makes it false, then the result is U. If, however, the result of the relationship can be resolved unambiguously for all combinations of U's, then the result is a 1 or 0. Z operand values are treated as U's.
Expression Operators

Suggestions

For a multibit compare, use EQL. The equal (=) sign cannot be used in place of the EQL operator, because it is used to assign values.

Examples

1. SIM> EVAL $#2 EQL(QOITE) $#2
   1
2. SIM> EVAL U#2 NEQ(QOITE) U#2
   0
3. SIM> EVAL U#2 EQL U#2
   U
4. SIM> EVAL 1U#2 GTR U0#2
   U
5. SIM> EVAL 1U#2 GEQ U0#2
   1

D.17 Shift Operators

The shift operators shift the first operand by the amount indicated in the second operand.

Application

Command Language, Behavior Language

Format

operand1 SL0 [ (directive) ] operand2
operand1 SL1 [ (directive) ] operand2
operand1 SLR [ (directive) ] operand2
operand1 SR0 [ (directive) ] operand2
operand1 SR1 [ (directive) ] operand2
operand1 SRR [ (directive) ] operand2

Description

The shift operators are abbreviations. The first character (S) stands for shift, the second character stands for the direction of the shift, left (L) or right (R), the third for what is to be shifted in: zeros (0), ones (1), or rotated bits (R).

The second operand indicates the amount by which the operand is to be shifted. If the second operand is greater than the width of the first operand, the result is all 0's, 1's, or rotated (R) bits, depending on the designation in the operator.
Table D–4  Shift Operator Definitions

<table>
<thead>
<tr>
<th>Shift Operand</th>
<th>Direction</th>
<th>Shift In</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL0</td>
<td>left</td>
<td>0</td>
</tr>
<tr>
<td>SR0</td>
<td>right</td>
<td>0</td>
</tr>
<tr>
<td>SL1</td>
<td>left</td>
<td>1</td>
</tr>
<tr>
<td>SR1</td>
<td>right</td>
<td>1</td>
</tr>
<tr>
<td>SLR</td>
<td>left</td>
<td>MSB (rotated from MSB)</td>
</tr>
<tr>
<td>SRR</td>
<td>right</td>
<td>LSB (rotated from LSB)</td>
</tr>
</tbody>
</table>

Behavior Language Directives

(WIDTH=integer) specifies width of expression.

See Section D.3 for details.

Width of Result

Width of first operand.

Undefined Handling

In the Behavior Language, undefined bits in the second operand result in the lowest possible shiftcount. The bits in the result that could be either ones or zeros are assigned U's. See Example 8 below.

In the Command Language, undefined bits in the second operand give undetermined results.

In the Command Language and the Behavior Language, undefined bits and wildcards (?) in the first operand are shifted verbatim.

Suggestions

Since the width result is the width of the first operand, you sometimes have to add 0's on the most significant bit side of operand1 in order to get the desired result.

Example

```
SIM> RADIX 2
SIM> EVAL 1#2 SLO 1
       ! Result is 0 because the width of
       ! operand1 is 1.
0
SIM> EVAL 01#2 SLO 1
       ! The 0 in front of the operand
       ! gives correct answer.
10
```
Expression Operators

You can use concatenation instead of one of the shift operators. Note the results in the following example.

Examples

1. SIM> RADIX 16
   SIM> STATE REG <15:0>
   SIM> EVAL REG=5
   SIM> EVAL REG<4:0> SLO 1       ! Shift 5-bit value in REG by 1,
        A                               ! giving 5-bit result
   SIM> EVAL REG<12:16>             ! 5-bit result drops high-order
   SIM> EVAL REG<4:0> SLO 1         ! bit
        4
   SIM> EVAL (REG<4:0>)<31:0> SLO 1 ! Force full result
        24                             ! May be easier to use
        24                             ! concatenation
        24                             ! Has same effect as SLO 1

2. SIM> EVAL 1010101#2 SRR 2       !Command Language
   110101

3. SIM> EVAL 111100#2 SLR 3        !Behavior Language
   100111

4. SIM> EVAL 1#2 SLO 0
   1
   SIM> EVAL 1#2 SLO 1
   0
   SIM> EVAL 01#2 SLO 1
   10

5. a SLO 1
   where a<7:0> = 1010#2
   Result: 0100#2

6. REG = REG<14:0> & CARRY_IN;    !Splice in a
                                  !variable

The example above shows that you can shift a variable by using the concatenation operator.

7. REG = REG<15:0> SLL 1;         !Shift in a
   where REG = 1010101010101010#2
   Result: 0101010101010101#2

8. A SLO B
   where A = 00000111#2 and B = U#2
   Result: 0U#16 or 0000UUUU#2

D.18 Tristate Operators

There are three tristate operators: TS, MOSTS, amd TCON.

Application

Command Language, Behavior Language, Network Equation statements.
For fourstate data types only.
Expression Operators

Format

data TS enable
source MOSTS gate
operand1 TSCON operand2

Description

The tristate (TS) operator facilitates modeling a tristate driver. The operator accepts two inputs: data and enable. When enable is high (1), the data is passed to the output; when enable is low (0), the output reflects a disconnected line, which is represented by the Z state.

The MOS tristate (MOSTS) operator is used in MOS 4-state expressions.

Both MOSTS and TS must have a single-bit right hand operand.

The tristate consensus (TSCON) operator computes the tristate consensus function, modeling a tristate bus. TSCON is used in conjunction with TS and MOSTS drivers.

Behavior Language Directives

None

Width of Result

The following table shows the resulting widths for the tristate operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Width of Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS</td>
<td>width of “data”</td>
</tr>
<tr>
<td>MOSTS</td>
<td>width of “source”</td>
</tr>
<tr>
<td>TSCON</td>
<td>width of the wider operand</td>
</tr>
</tbody>
</table>

Truth Tables

Table D–5 TS Truth Table

<table>
<thead>
<tr>
<th>Data</th>
<th>Enable</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1,Z,U</td>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>0,1,Z,U</td>
<td>1</td>
<td>0,1,U,U</td>
</tr>
<tr>
<td>0,1,Z,U</td>
<td>Z,U</td>
<td>U,U,U,U</td>
</tr>
</tbody>
</table>

D–36
Table D–6  MOSTS Truth Table

<table>
<thead>
<tr>
<th>Source</th>
<th>Gate</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1,Z,U</td>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>0,1,Z,U</td>
<td>1</td>
<td>0,1,Z,U</td>
</tr>
<tr>
<td>0,1,Z,U</td>
<td>Z,U</td>
<td>U,U,Z,U</td>
</tr>
</tbody>
</table>

Table D–7  TCON Truth Table

<table>
<thead>
<tr>
<th>Operand1</th>
<th>Operand2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1,Z,U</td>
<td>0</td>
<td>0,U,0,U</td>
</tr>
<tr>
<td>0,1,Z,U</td>
<td>1</td>
<td>U,1,1,U</td>
</tr>
<tr>
<td>0,1,Z,U</td>
<td>Z</td>
<td>0,1,Z,U</td>
</tr>
<tr>
<td>0,1,Z,U</td>
<td>U</td>
<td>U,U,U,U</td>
</tr>
</tbody>
</table>
ZYCAD Partitions

You can improve ZYCAD simulation performance by partitioning your network into sets of gates with comparable amounts of gate activity. You then assign these sets, or partitions, to particular ZYCAD S-modules.

To understand how to optimize ZYCAD simulation performance by creating partitions, you need some understanding of the ZYCAD Logic Evaluator architecture.

E.1 ZYCAD Logic Evaluator Architecture

The ZYCAD Logic Evaluator is made up of one M-Module (memory) and one to sixteen S-Modules (simulation), all of which communicate over a global communication bus.

The M-Module controls the advancing of the simulation time clock, application of pattern stimuli, read/writes of the functional memory planes, and communication with the host computer. It has the top priority in accessing the global communication bus.

Each S-Module performs the logic simulation on the gates residing in it. All the S-Modules simulate in parallel for each time slot. Priority in accessing the global communication bus depends on how close the S-module is to the M-module; the closest S-module has the highest priority, the most distant S-module has the lowest priority.

The global communication bus is used for:

- Communication between S-modules. Fanout propagation can cause a driver in one S-module to send a logic value over the bus to its fanout in another S-module.

- Functional memory activity. When a memory input line changes, the value is sent over the global communication bus to the M-module, where it is stored. Likewise, when a memory read is performed, the value is sent from the M-module over the bus to the pertinent S-modules.
E.1.1 Bottlenecks

If you do not specify how to partition, the DECSIM/ZYCAD interface distributes your network among the S-modules, each of which can hold 64K gates.

Since the S-modules simulate in parallel, one time slot at a time, and since no S-module can proceed to another time slot until all S-modules have completed the activity in the current time slot, heavy activity in one S-module can cause more lightly loaded S-modules to sit idle.

In the same vein, since all S-modules communicate both with each other and with the M-module over the same global communication bus, heavy communication in one module can deny lower-priority S-modules access to the global communication bus.

An S-Module is forced to discontinue simulation during a time slot when any of the following occurs:

- It has not received data from a functional memory read that should have matured during the present time slot.
- It has tried to communicate over the global bus, but cannot access it because another module with a higher priority has it.
- It has completed its simulation for a time slot, but cannot proceed to the next time slot because another S-module is still simulating.

You try to minimize the above occurrences by distributing network activity equally and minimizing communication between S-modules. This produces the most simulation events per second.

E.1.2 How the DECSIM/ZYCAD Interface Partitions a Network

Because an S-module can hold only 64K gates, all networks containing more than 64K gates must be partitioned.

If you do not specify how to partition, the DECSIM/ZYCAD interface uses your modeling hierarchy as a guide to distribute the network among the S-modules. This ensures that models are not arbitrarily split apart; it does not ensure equal activity in each S-module, nor does it minimize communication between S-modules.

Whether you specify partitioning or not, the DECSIM/ZYCAD interface creates some partitions according to the following Wiretie Partition Rule.

- All drivers of a phantom (ZYCAD node model) must reside in the same S-Module as the phantom.

When the DECSIM/ZYCAD Interface produces the ZYCAD network representation, it determines which gates violate the above rule. Drivers of the same phantom are then locked together to eliminate the rule violation. Each set of these gates is called a "hard partition." Hard partitions are the first partitions allocated to S-Modules.
E.2 How to Partition

You specify how your network is partitioned by writing a partition input file (PAI), and pointing to it with the COMPIL/PARTITION_INPUT or USE/PARTITION_INPUT commands.

The partition output file (PAO) reports how your network was partitioned. It uses the same format as the PAI file, which enables you to modify it and use it as a PAI file.

The following syntax applies to both files.

E.2.1 Syntax of Partition Input (PAI) File

The PAI file tells the DECSIM/ZYCAD interface how to distribute a network among the available S-modules. There are three commands that appear in a PAI file.

- **ZYCAD_S_MODULES** specifies how many S-modules should be used in the simulation.
- **ZYCAD_PARTITION** specifies a set of instantiated models. For example, **ZYCAD_PARTITION 1 = E3.G1** groups all models in instance E3.G1 into a set called "partition 1."
- **ZYCAD_S_MAP** specifies which partitions are placed in which S-modules. For example, **ZYCAD_S_MAP 3 = 1** places partition 1 into S-module 3.

You can use the standard DECSIM comment (!) and line continuation conventions in a PAI file. Comments placed in the PAI file are not echoed in the PAO file.

The three PAI file commands are described in detail below:

**ZYCAD_S_MODULES = integer;**

Specifies how many S-modules to use in the simulation. You can specify fewer S-modules than are available in the LE, but you cannot specify more. This value is compared to that assigned ZYCAD$S_MODULES at installation time.

**Example**

```plaintext
ZYCAD_S_MODULES = 4;
```

**ZYCAD_PARTITION integer_part = label [,label,label, ...]**

Specifies a partition containing any number of model instantiations and identifies it with an integer, integer_part. A partition is defined as a set of instantiated models that reside in the same S-module during simulation.

You must specify the model hierarchy with labels from the network level down. The labels that make up a partition can each be considered a "parent"; all its "children" (lower level models) are included in the same partition. If an instance is not labeled, you cannot govern how it is partitioned. The DECSIM/ZYCAD interface arbitrarily partitions it, but the PAO file does not report its location.
ZYCAD Partitions

No two partitions can overlap.

Any number of ZYCAD_PARTITION lines can reside in an input partition file.

The location and partition number of a specific ZYCAD_PARTITION line in your file affects its chances of being placed in the specified S-module (sometimes a partition may not fit). The partitions that are listed first in the file, and that have the lowest partition numbers, are placed in S-modules first.

All ZYCAD_PARTITION commands must appear prior to any ZYCAD_S_MAP commands.

In the following example, the user specified that two portions of his network be partitioned, and allowed the interface to partition the rest of his network.

Example

ZYCAD_PARTITION 1 = E3.G2; ! User Partition
ZYCAD_PARTITION 2 = E2; ! User Partition

ZYCAD_S_MAP integer_smapi = integer_part [=,integer,integer,...];

Specifies which partitions are placed in which S-module. S-modules are identified with an integer, integer_smapi. Any number of ZYCAD_PARTITION numbers can appear on the same line.

In the following example, the user assigned partitions 2 and 1 to S-modules 3 and 2.

Example

ZYCAD_S_MAP 3 = 2;
ZYCAD_S_MAP 2 = 1;

E.2.2 Sample Partition Output Files

The partition output file reports:

- Partitions specified in the PAI file
- Partitions created by the DECSIM/ZYCAD interface that have user-specified labels. If you do not label instances, they are not listed.

The following examples are two partition output files and a partition input file. The first example shows an output file that reports partitioning generated only by the DECSIM/ZYCAD interface. The second example shows a partition input file and the resulting partition output file.
Example 1

```plaintext
!++
! Partition Output File : DSUS\LTSMITH\PART1.PAO;91
! For Network File : DSUS\LTSMITH\PART1.NOB;12

ZYCAD_PARTITION 1 = E1;
ZYCAD_PARTITION 2 = E2;
ZYCAD_PARTITION 3 = E3;
ZYCAD_PARTITION 4 = E4;

ZYCAD_S MODULES = 4;

!++
! No partition rules checked since network is less than 64K gates
! and no Partition Input File (FAI) was specified.

! Total Number of Gates Used : 16

! End of Partition Output File
```

Example 2

```plaintext
!++
! Partition Input File for test network PART1.NOB;12
! 
ZYCAD_PARTITION 1 = E3.G2;
ZYCAD_PARTITION 2 = E2;

ZYCAD_S MODULES = 4;
ZYCAD_S MAP 1 = 3;
ZYCAD_S MAP 2 = 1;

! End of Partition Input File

!++
! Partition Output File : DSUS\LTSMITH\TPART1.PAO;17
! For Network File : DSUS\LTSMITH\PART1.NOB;12
! DEC S I M Version V4.1-1774 26-DEC-1984 17:04:02

ZYCAD_PARTITION 1 = E3.G2;
ZYCAD_PARTITION 2 = E2;
ZYCAD_PARTITION 3 = E1;
ZYCAD_PARTITION 4 = E3.G1;
ZYCAD_PARTITION 5 = E4;

ZYCAD_S MODULES = 4;

ZYCAD_S MAP 1 = 3;
ZYCAD_S MAP 3 = 2;
ZYCAD_S MAP 1 = 4;
ZYCAD_S MAP 2 = 1;
ZYCAD_S MAP 1 = 5;

!++
! Gates per S Module :
! S Module 1 : 11
! S Module 2 : 1
! S Module 3 : 4
! S Module 4 : 0

! Total Number of Gates Used : 16

! End of Partition Output File
```
E.2.3 Where Partitioning Can Fail

If the logical name ZYCAD$S_MODULES is not defined correctly and no PAI file is specified, then the test determining whether the total gates will fit in the available S-Modules will fail. You will receive the error message:

%W-NTL Network too large to fit into the following number of S-Modules - xx
%E-ZPE ZYCAD Partitioning Error; Please contact DECSIM/ZYCAD Support.

This error will also be seen if the total number of gates in the network exceeds the capacity of the total S-Modules used.

The DECSIM/ZYCAD interface creates hard partitions according to the Wiretie Partitioning Rule. Each hard partition can occupy only one S-module; consequently, if it contains more than 64K gates, it does not fit. Also, several small hard partitions can take up so much space in S-modules that an additional hard partition may not fit, even if it has less than 64K gates. The DECSIM/ZYCAD support team assists design teams in resolving such problems.

E.2.4 Tuning Suggestions for Large Networks

Before partitioning a large network, compile and simulate it without a PAI file, and assess its performance.

Next, partition it just as the design was partitioned among individual designers.

Compare the gate sizes per S-module found in the PAO files with the S-module event counts in the ZOF file. Determine which S-modules have the most activity. Then, transfer the partitions in those modules to the first and last S-modules, and compare again. This provides data points for the speed of the ZYCAD global bus and the activity of the gates across the S-modules.
Process Parameter File Format

The MOS process parameter file allows a design team to define parameters for a specific process such as ZMOS or CMOS, or to change the parameters for specific circuit cases such as a DLATCH feedback transistor. The process parameter file is typically set up at the beginning of a project based on SPICE experiments and other project files, then used throughout that project.

Since all resistances and capacitances used in DECSIM MOS simulation are calculated using this file, it is crucial that the process parameter file is set up properly for the project, and that the implications of its setup are fully understood by the members of the project. The process parameter file provides the foundation for the accuracy of DECSIM MOS simulation.

DECSIM uses the process parameter file during compilation to calculate the resistances and capacitances for each MOSFET in a circuit. The file is given the length, width, and type of the MOSFET.

The following figure shows how the process parameter file is used during compilation:

```
+-----------------+    +-----------------+    +-----------------+
| PROCESS PARAMETER| ----> | Compiler         | ----> | Calculated MOSFET |
| FILE             |        |                    |        |                    |
+-----------------+    +-----------------+    +-----------------+
| Supplied MOSFET  | ----> | Calculation      |        |                    |
| data:            |        |                    |        |                    |
|    o Model type  |        |                    |        |                    |
|    o Length      |        |                    |        |                    |
|    o Width       |        |                    |        |                    |
|    o Usage (PULLUP, PULLDOWN, etc) | |                    |        |                    |
```

The process parameter file is a readable file that is set up in four main sections. Each section is headed by a keyword, followed by a colon. The sections must appear in the following order:

1. PARAMETER_SECTION
2. PARASITIC_PARAMETER
3. RESISTANCE_SECTION
4. DEVICE_SECTION

End--of--line comments are allowed in each section and are indicated by the exclamation character (!).

These four sections of the process parameter file are described in detail in the remainder of this appendix.
F.1 PARAMETER_SECTION

The PARAMETER_SECTION contains coefficients and default values in the form of parameter definitions. Every parameter definition must be specified in this section; however, the order can be arbitrary. You must end a parameter definition with a semicolon.

The following is an example of the PARAMETER_SECTION.

Example

PARAMETER_SECTION:
    IDENT = 'Decsim Standard ZMOS, Rev 1';
    LOWTHRESH = 0.2;
    HIGHTHRESH = 0.8;
    DEFAULT_TIMEOUT = INFINITY;
    MAXRES = INFINITY;
    CAPGA = 1000 UF;
    CHAIN_INTER_CONTRIB = FALSE;
    NODE_CAPAC_DEFAULT = 10 FF;

F.1.1 IDENT

The IDENT parameter identifies the file and records the revision number. The IDENT string, delimited by single quotes, allows you to check that the correct process parameter file is being used to compile, as this string is printed when the network is compiled. The syntax for the IDENT parameter is:

IDENT = 'identification string in single quotes';

F.1.2 LOWTHRESH and HIGHTHRESH

The LOWTHRESH and HIGHTHRESH parameters supply the default node threshold values. All node thresholds are set to these values unless they are specifically overridden with the THRESHOLD node attribute on a per node basis.

Both threshold values are normalized voltages between 0.0 and 1.0. The HIGHTHRESH value must be greater than or equal to the LOWTHRESH value. The following is an example of the LOWTHRESH and HIGHTHRESH parameters:

Example

LOWTHRESH = 0.2;
HIGHTHRESH = 0.8;
F.1.3 DEFAULT_TIMEOUT

The DEFAULT_TIMEOUT parameter supplies the default node timeout value. All node timeouts are set to this value except in the following situations:

- The node timeout value is specifically overridden on a per-node basis with the TIMEOUT node attribute.
- The DEFAULT_TIMEOUT value is overridden by the DEFAULT_TIMEOUT network statement.

Valid timeout values are from 0.0 to INFINITY, specified in seconds.

F.1.4 MAXRES

The MAXRES parameter supplies a resistance value for the entire circuit. A value greater than this value causes DECSIM to consider a MOSFET off or open for computing capacitance at that node. A default circuit MAXRES value can be overridden by the MAXRES network statement.

Valid MAXRES resistance values are from 0.0 to INFINITY, specified in ohms. Setting MAXRES to 0.0 turns off charge-sharing calculations but causes the simulator delay calculation to lose accuracy.

F.1.5 Nonparasitic Capacitance Coefficients and Controls

The compiler's capacitance calculations compute a single capacitance for every node in your circuit. The coefficients for these calculations can be parasitic or nonparasitic.

The following process file parameters serve as nonparasitic coefficients or control variables in the compiler's capacitance calculations:

- CAPGA. CAPGA is the default gate area capacitance coefficient in units of capacitance per square-meter. This value is specified in microfarads. Gate capacitance is calculated as a function of width, length, and CAPGA. If CAPGA is not specified in the PARAMETER_SECTION, it must be specified in the PARASITIC_PARAMETER section for each process used.

- CHAIN_INTER_CONTRIB = TRUE or FALSE. CHAIN_INTER_CONTRIB indicates if capacitance that is internal to a stack contributes to the capacitance of the node that is at the top of the stack.

- NODE_CAPAC_DEFAULT. NODE_CAPAC_DEFAULT is the default fixed capacitance for a node, specified in microfarads. It is added to every node's capacitance unless explicit fixed capacitance is supplied with a CAPACITANCE statement. This fixed capacitance supplies an estimate for inter-connect (metal) capacitance.

The coefficients used to compute the parasitic capacitance are defined in the PARASITIC_PARAMETER section described in Section F.2.
F.1.6 Calculating the Node Capacitance

The following steps describe how the compiler calculates the node capacitance:

1. The following quantities are summed to produce the capacitance of a node:
   - Node Fixed-capacitance
   - Gate-capacitance
   - Parasitic-capacitance
   - Stack-inter-capacitance

2. The node fixed-capacitance is supplied either explicitly by the CAPACITANCE declaration, or, if there is no explicit specification, by the NODE_CAPAC_DEFAULT parameter value.

3. The gate-capacitance is supplied by the sum of the capacitance of every MOSFET gate terminal that connects to the node. The exception is when MOSFET is connected as a LOAD or DIODE. The gate-capacitance is calculated by the following equation:

   \[ \text{Gate-capacitance (F)} = \text{width (m)} \times \text{length (m)} \times \text{CAPGA (F/sq-m)} \]

4. The parasitic-capacitance is supplied by either PARASITIC declaration area and periphery for the node, or, if there is no PARASITIC area and periphery specified, by summing the capacitance of every MOSFET source or drain connected to the node.

   If there is a PARASITIC declaration, then the following equation is used to calculate the parasitic capacitance:

   \[
   \text{parasitic-capacitance (F)} = \text{CAPDA (F/sq-m)} \times \text{PARASITIC-area (sq-m)} + \text{CAPDP (F/m)} \times \text{PARASITIC-periphery (m)}
   \]

   The PARASITIC-area and PARASITIC-periphery values are specified by the PARASITIC declaration.

   If there is no PARASITIC declaration, then the following equation is used to calculate the parasitic capacitance:

   \[
   \text{parasitic-capacitance (F)} = \text{CAPDA (F/sq-m)} \times \text{source-drain-area (sq-m)} + \text{CAPDP (F/m)} \times \text{source-drain-periphery (m)}
   \]

   The source-drain-area and source-drain-periphery are calculated using the transistor-width and the Area and Periphery coefficients.
F.2

**PARASITIC PARAMETER**

The PARASITIC PARAMETER section contains a number of coefficient definitions that can appear in an arbitrary order. However, every coefficient definition must be specified. End each parameter definition with a semicolon.

Unlike other sections, you can write more than one PARASITIC PARAMETER section. Each PARASITIC PARAMETER section is identified by a parasitic-parameter-type name. Multiple PARASITIC PARAMETER sections in a process parameter file allow different sets of coefficients to compute capacitance. The coefficients must be either the area and periphery from a PARASITIC or PCAP declaration in the wirelist, or the dimensions of the sources and drains that connect to that node.

Section F.4, which defines MOSFET devices, contains the names of each PARASITIC PARAMETER section, associated with a particular device type. This directs NETPRO to calculate the parasitic capacitance for that particular MOSFET device using the parameters that appear in a specific PARASITIC PARAMETER section.

The following groups of coefficients are defined in a PARASITIC PARAMETER section:

- Area coefficients: compute source/drain area using a transistor width value.
- Periphery coefficients: compute source/drain periphery using a transistor width value.
- Parasitic capacitance coefficients: compute parasitic capacitance using source/drain area and periphery values.

The following is an example of a PARASITIC PARAMETER section named TEST1.

```
PARASITIC PARAMETER TEST1:
AREA_WIDTH_MULTIPLIER = 4 UM;
AREA_CONSTANT = 45 SQ UM;
MIN_AREA = 45 SQ UM;
PERIPH_WIDTH_MULTIPLIER = 2;
PERIPH_CONSTANT = 18 UM;
MIN_PERIPH = 36 UM;
CAPDA = 1000 UF;
CAPDP = 0.01 UF;
```

F.2.1 **Area and Periphery Coefficients**

The area and periphery coefficients calculate the area and periphery of a MOSFET source or drain, based on the width of that source or drain.

The following process file parameters serve as coefficients in the compiler's area and periphery calculations. The parameter definitions should be based on the source/drain geometry that appears in the following figure.
The syntax for each parameter is KEYWORD = value, where value is a number in optional units. The following parameter keywords show how they are calculated:

- AREA_WIDTH_MULTIPLIER = "e" dimension from geometry above. The value is a distance, specified in microns.
- AREA_CONSTANT = c * b. The value is an area, specified in microns.
- MIN_AREA = a * b. The value is an area, specified in microns.
- PERIPH_WIDTH_MULTIPLIER = 1 or 2, depending on whether you want to count the width once or twice in the periphery. This value has no units.
- PERIPH_CONSTANT = 2 * a. The value is a distance, specified in microns.
- MIN_PERIPH = (2 * a) + (2 * b). The value is a distance, specified in microns.

The compiler calculates source/drain periphery and area as follows:

Area (sq-m) =
\[
\text{MAX (AREA_WIDTH_MULTIPLIER} \times \text{width (m)} + \text{AREA_CONSTANT (sq-m),}
\]
\[
\text{MIN_AREA (sq-m)})
\]

Periphery (m) =
\[
\text{MAX (PERIPH_WIDTH_MULTIPLIER} \times \text{width (m)} + \text{PERIPH_CONSTANT (m),}
\]
\[
\text{MIN_PERIPH (m))}
\]
F.2.2 Parasitic Capacitance Coefficients

The parasitic capacitance coefficients compute parasitic capacitance given source or drain area and periphery. You can specify the following as parasitic capacitance coefficients:

- **CAPGA.** CAPGA must be specified here or in the nonparasitic section (the PARAMETER_SECTION). If it is not specified here, CAPGA defaults to the nonparasitic section value.
- **CAPDA.** Diffusion area (bottom–wall) capacitance coefficient, in capacitance per sq–meter. Default units are pico–farads (PF) per sq–micron.
- **CAPDP.** Diffusion periphery (side–wall) capacitance coefficient, in capacitance per meter. Default units are pico–farads (PF) per meter.
- **DELTA_L, DELTA_W.** Gate shrinkage. During fabrication, the gate width and length can shrink by a small amount from the drawn values. DELTA_W and DELTA_L are subtracted from the MOSFET width and length in the determination of node capacitances and resistances. The default value is 0, and the default unit is microns.

F.3 RESISTANCE_SECTION

The RESISTANCE_SECTION defines the resistance values for different MOSFETs of different sizes. These values are typically derived from SPICE simulations.

Each definition in the RESISTANCE_SECTION sets equal a resistance–set–name and the data points that define that resistance–set. These resistance–sets are then referenced in the DEVICE_SECTION that is described in Section F.4. All resistance–sets referenced in the DEVICE_SECTION must be defined in the RESISTANCE_SECTION.

Note: The resistance values specified in this section are in units of OHMS, not ohms–per–square.

F.3.1 Interpolation of Resistance Values

If any MOSFETs appear in your circuit where the MOSFET length and width do not match the length and width of a given data point, the compiler interpolates the appropriate resistance values for that MOSFET.

Interpolation is done through the following steps:

1. Convert the specified resistance in ohms to ohms–per–square units by multiplying by width/length.
2. Interpolate using the ohms–per–square unit.
3. Convert back to ohms by multiplying by length/width.
The following is an example of the RESISTANCE_SECTION:

RESISTANCE_SECTION:

\[
\begin{align*}
! \text{Definition} \ # \ 1 \ & \text{defining resistance-set } "R\_N" \\
R\_N \ &= \ (10 \ \text{UM} \ 10 \ \text{UM} \quad 122.153 \ \text{K} \quad 35.000 \ \text{K} \quad 43.333 \ \text{K}); \\

! \text{Definition} \ # \ 2 \ & \text{defining resistance-set } "R\_D" \\
R\_D \ &= \ (22 \ \text{UM} \ 8 \ \text{UM} \quad 82.061 \ \text{K} \quad 6.541 \ \text{K} \quad 15.203 \ \text{K}) \\
&\quad (17 \ \text{UM} \ 8 \ \text{UM} \quad 116.052 \ \text{K} \quad 9.250 \ \text{K} \quad 21.500 \ \text{K}) \\
&\quad (14 \ \text{UM} \ 8 \ \text{UM} \quad 164.122 \ \text{K} \quad 13.082 \ \text{K} \quad 30.406 \ \text{K}) \\
&\quad (12 \ \text{UM} \ 8 \ \text{UM} \quad 232.104 \ \text{K} \quad 18.500 \ \text{K} \quad 43.000 \ \text{K}) \\
&\quad (10 \ \text{UM} \ 8 \ \text{UM} \quad 328.244 \ \text{K} \quad 26.163 \ \text{K} \quad 60.811 \ \text{K}) \\
&\quad (8 \ \text{UM} \ 8 \ \text{UM} \quad 464.208 \ \text{K} \quad 37.000 \ \text{K} \quad 86.000 \ \text{K}) \\
&\quad (8 \ \text{UM} \ 10 \ \text{UM} \quad 656.489 \ \text{K} \quad 52.326 \ \text{K} \quad 121.622 \ \text{K}) \\
&\quad (8 \ \text{UM} \ 13 \ \text{UM} \quad 928.415 \ \text{K} \quad 74.000 \ \text{K} \quad 172.000 \ \text{K}) \\
&\quad (8 \ \text{UM} \ 17 \ \text{UM} \quad 1312.978 \ \text{K} \quad 104.652 \ \text{K} \quad 243.244 \ \text{K}); \\
\end{align*}
\]

F.3.2 RESISTANCE_SECTION Syntax

The RESISTANCE_SECTION is a collection of definitions. The following shows the syntax for each definition:

```
resistance-set-name = (width-1
  length-1
  static-resistance-1
  dynamic-high-resistance-1
  dynamic-low-resistance-1)

  (width-2
  length-2
  static-resistance-2
  dynamic-high-resistance-2
  dynamic-low-resistance-2)

  ......  

  ......  

  ......  

  (width-n
  length-n
  static-resistance-n
  dynamic-high-resistance-n
  dynamic-low-resistance-n);
```

Each definition must contain at least one data-point (surrounded by parentheses), but can contain an unlimited number. End each definition with a semi-colon.

All length−width combinations specified for a resistance-set must be unique. Width and length are distances, specified in microns. Resistance is specified in ohms.

The static resistance calculates the state−value during node evaluation in the simulator. The dynamic resistances calculate the delay−value during node evaluation. In some cases, you can leave the dynamic−high−resistance or the dynamic−low−resistance unspecified for a data point.

For example, a resistance-set that is used only for pullup devices does not need the dynamic−low−resistance specified. Instead, supply the string "_*__
" for the dynamic resistance value. Note that the static−resistance must always be supplied. The following example shows the format:

Example

```
(10 \text{ UM} \ 8 \text{ UM} \ 328.244 \text{ K} \ 26.163 \text{ K} \ _*_ * ) \\
```

!dyn-low is omitted
F.4 DEVICE_SECTION

The DEVICE SECTION describes how to translate wirelisted MOSFETs into DECSIM MOS models with specified resistance values. This translation is established by a set of definitions.

Each definition sets equal a device-name and that device's modeling-data, as used in different circuits. The device-name string is the name that appears in the TYPE parameter of the MOSFET statement in the MOS Model Library.

The following is an example of the DEVICE SECTION:

Example

DEVICE_SECTION:

! Definition # 1
N (N_PARASITIC) =
[ PULLUP]: (N (R_N), ZMOS_N_PULLUP)
[ PULLUP_DIOGE]: (R (R_N), ZMOS_N_PULLUP_DIOGE)
[ PULLUP_LOAD]: (Z () , ZMOS_N_PULLUP_LOAD)
[ PULLDOWN]: (N (R_N), ZMOS_N_PULLDOWN)
[ PULLDOWN_LOAD]: (Z () , ZMOS_N_PULLDOWN_LOAD)
[ PULLDOWN_DIOGE]: (R (R_N), ZMOS_N_PULLDOWN_DIOGE)
[ TRANSMISSION]: (N (R_N), ZMOS_N_TRANSMISSION)
[ DIODE]: (N (R_N), ZMOS_N_DIODE)
[ CAPACITOR]: (Z () , ZMOS_N_CAPACITOR);

! Definition # 2
D (D_PARASITIC) =
[ PULLUP]: (R (R_D), ZMOS_D_PULLUP)
[ PULLUP_DIOGE]: (R (R_D), ZMOS_D_PULLUP_DIOGE)
[ PULLUP_LOAD]: (R (R_D_UP_LOAD), ZMOS_D_PULLUP_LOAD)
[ PULLDOWN]: (R (R_D), ZMOS_D_PULLDOWN)
[ PULLDOWN_LOAD]: (R (R_D), ZMOS_D_PULLDOWN_LOAD)
[ PULLDOWN_DIOGE]: (R (R_D), ZMOS_D_PULLDOWN_DIOGE)
[ TRANSMISSION]: (R (R_D), ZMOS_D_TRANSMISSION)
[ DIODE]: (R (R_D), ZMOS_D_DIODE)
[ CAPACITOR]: (Z () , ZMOS_D_CAPACITOR);

F.4.1 DEVICE_SECTION Syntax

The definitions in the DEVICE SECTION have the following syntax:

Device-name (Parasitic-parameter-type name) =

[ Case-keyword-1 ] : ( DECSIM-primitive-MOS-model-type-1
( Resistance-set-name-1 ),
 usage-name-1)

[ Case-keyword-2 ] : ( DECSIM-primitive-MOS-model-type-2
( Resistance-set-name-2 ),
 usage-name-2)

......

[ Case-keyword-n ] : ( DECSIM-primitive-MOS-model-type-n
( Resistance-set-name-n ),
 usage-name-n);
F.4.2 Parasitic–Parameter–Type Name

The parasitic–parameter–type name points to a PARASITIC_PARAMETER section, defined previously in the process parameter file. See Section F.2. This name specifies the PARASITIC_PARAMETER section coefficients used to calculate source or drain capacitance.

F.4.3 Case Keywords

The case keywords define not only the device, but how the device is connected in the circuit. For example, a "D" MOSFET connected as a pullup–load can be modeled completely differently from a "D" MOSFET connected in some other way. Every case keyword must appear in every device definition.

Valid case keywords that you can specify are:

- PULLUP – connected in a path between a node and high–power. It is modeled unidirectionally.
- PULLUP_DIODE – connected in a path between a node and high–power. The MOSFET's gate terminal is connected toward high–power. It is modeled unidirectionally.
- PULLUP_LOAD – connected in a path between a node and high–power. The MOSFET's gate terminal is connected toward the node. It is modeled unidirectionally.
- PULLDOWN – connected in a path between a node and low–power. It is modeled unidirectionally.
- PULLDOWN_DIODE – connected in a path between a node and low–power. The MOSFET's gate terminal is connected toward the node. It is modeled unidirectionally.
- PULLDOWN_LOAD – connected in a path between a node and low–power. The MOSFET's gate terminal is connected toward low–power. It is modeled unidirectionally.
- TRANSMISSION – connected in a path between two nodes. It is modeled bidirectionally.
- DIODE – connected in a path between two nodes. The MOSFET's gate terminal is connected to one of these nodes. It is modeled bidirectionally.
- CAPACITOR – source and drain of the MOSFET are connected together.
F.4.4  **DECSIM Primitive MOS Model Types**

Specify the DECSIM-primitive-MOS-model-type in the DEVICE definition to indicate the primitive type that DECSIM uses to model the device. You can specify six primitive MOS model types:

- N–ON if input is 1; OFF if input is 0
- P–ON if input is 0; OFF if input is 1
- D–ON STRONG if input is 1; ON WEAK if input is 0
- DI–ON STRONG if input is 0; ON WEAK if input is 1
- R–Not affected by input, always ON
- Z–Never conducts, always OFF.

**Note:** Do not confuse DECSIM-primitive-MOS-model-type with the user-supplied device name. They are not the same thing.

F.4.5  **Resistance–Set–Names**

Specifying the resistance–set–names in the DEVICE definition indicates the resistance–sets that DECSIM associates with the device. You must define the resistance–set–names in the RESISTANCE_SECTION before you specify them in the DEVICE_SECTION definitions.

The DECSIM-primitive-MOS-model-types above have 0, 1, or 2 resistance–sets associated with them. The syntax resembles the MOS–model–type with the 0, 1, or 2 resistance–set–names in parentheses, separated by commas. The following are the syntax formats:

- N–(1 resistance set) N (res–set)
- P–(1 resistance set) P (res–set)
- R–(1 resistance set) R (res–set)
- Z–(0 resistance sets) Z ()

F.4.6  **Usage Names**

A usage name is a label for a transistor. It identifies transistors when you invoke the EXAMINE/FANIN/PARAM DECSIM command in the DECSIM command language.
Binary Trace File Format

The TRACE/BINARY and TRACE/REPLAYABLE commands make a high speed, high density capture of the trace information produced by the DECSIM simulator. Binary files can be used with the REPLAY command to display the trace information from a previous simulation. The files can also be used with the PATTERN/BINARY command as pattern files to drive a network.

This appendix explains how to create and read binary trace files, and how to extract data from them. To use the REPLAY or PATTERN/BINARY commands, you don't need to know how to read or extract data from binary trace files. The information in those sections is intended for people who want to use binary trace files as input to postprocessor applications programs, such as waveform display programs.

Other manuals that may be useful are: Introduction to VAX/VMS System Routines, VAX/VMS System Service Reference Manual, Guide to VAX/VMS File Applications.

G.1 Creating Binary Trace Files

When you elect to create a binary trace file, your choice of qualifiers on the TRACE and SET LOG commands are limited. The restrictions are described below.

G.1.1 Optional and Required Qualifiers on the TRACE Command

To create a binary trace file with the TRACE command, you must use the /LOGNAME qualifier and either the /BINARY or /REPLAY qualifier. The /REPLAY qualifier creates a vector format trace. The /BINARY qualifier also creates a vector format trace by default, although you may specify the normal format if you want. You may not use the odometer or graph formats in a binary trace file.

A vector trace captures the states of all the nodes in the access list whenever any one of those nodes transitions. In a normal trace, the state of an individual node is reported only when it transitions. See the TRACE command description for more information on the difference between normal and vector traces.

If the signal names in the trace access list do not match the signal names in the network to which you intend to input the file, you may use the /MAP qualifier. See the TRACE command description for more information on this qualifier.
G.1.2 Opening a Binary Trace File

The /LOGNAME qualifier on the TRACE command opens a trace file with the name you specify, if a log with that name is not currently open. If a log file with the specified name exists but is not open, the /LOGNAME qualifier opens a new version of that log file.

If you prefer, you may open a binary trace file with the SET LOG command. In this case, you must use the /TRACE and /NOHEADER qualifiers with the LOG command. You must also use the /LOGNAME qualifier with the TRACE command.

G.1.3 Concatenating Related Traces into a Single Trace File

To put related traces in the same trace file, you must execute the traces sequentially, not simultaneously. If you execute more than one trace in a single simulation session, the same node number will be assigned to more than one node and the data will be corrupted.

If you want to put data from a second or subsequent trace into a binary trace file that is currently open, simply specify that logname in the /LOGNAME qualifier on the TRACE command.

To put data from a trace into a binary trace file that is currently closed, execute a SET LOG /TRACE/NOHEADER/APPEND logname command and then specify that logname in the /LOGNAME qualifier on the TRACE command.

G.2 Reading Binary Trace Files

A binary trace file is a variable length, sequential file produced by VAX/VMS Record Management Services (RMS). The output from a single trace is called a binary trace module. As described above, you can concatenate the output from separate traces into a single binary trace file; thus a binary file may contain more than one module.

Each module consists of four sections, as described in Figure G–1.
Figure G–1  Binary Trace Module Format

<table>
<thead>
<tr>
<th>Module Header Records</th>
<th>Node Name Data Set</th>
<th>Node State Data Set</th>
<th>Module Trailer Records</th>
</tr>
</thead>
</table>

The Module Header Records contain information identifying the binary trace and describing how the trace was created. The Node Name Data Set contains the names of the nodes involved in the trace and information about the format of the Node State Data Set. The Node State Data Set is the group of records that describe the node transitions that occurred during the trace. The Module Trailer Records mark the end of the module.

Each of these sections contains records, whose number and size vary according to the section. (To save space, only the minimum number of bytes is used to store each record. The maximum record size is 32,763 bytes.) Each record is identified by class and type, and is further subdivided into fields, whose size and number vary according to the kind of data they store.

To read a binary trace file, exit from DECSIM and execute a `ANALYZE/RMS_FILE/INTERACTIVE binary_trace_filename` command. This command invokes a utility that dumps out the binary trace file in a tree structure. Using the utility’s `DOWN`, `UP`, `NEXT`, `BACK`, and `REST` commands you can navigate through the file, record by record. The records are in hexadecimal, with ASCII characters, if they occur, displayed in the far right column. The utility has online help, but you can also see *Guide to VAX/VMS File Applications* for more information.

The remainder of this section describes the trace module records, in the order in which they appear in the module. See Figure G–2 below for an overview of the record sequence. The acronyms in the figure refer to the class/type of each record.
G.2.1 Module Header Records (MHR)

The module header records (Class 1 records) identify the binary trace and describe how the trace was created. These records include the following types:

- Main Module Header (MMH)
- Module Description Record (MDR)
G.2.1.1 **Main Module Header (MMH)**

The Main Module Header record (Class 1, Type 1) is the first record found in any binary trace module. It must be preceded either by the beginning of the file or by the End Module Status record of another binary trace module. The presence of any other record type preceding this record is an error. This record must be followed by a Module Description Record.

The following example, generated from a binary trace module by the ANALYZE/RMS utility, illustrates the structure of the MMH record.

**DATA BYTES (VBN 1, offset %X'0000'):**

```
  7  6  5  4  3  2  1  0  1  1  2  3  4  5  6  7
-----------------------------------------
  00 00 00 02 01 01 00 1D 0000 ...........
  00 00 91 D1 2B A7 8E D6 0008 [0.5+8...]
  06 00 00 00 00 00 00 00 0010 ...........
  32 45 43 41 52 54 00 00 0018 [.TRACE2.]
```

**BYTES 0, 1 indicate the length of this record.**

**BYTE 2** indicates the class of this record. The field name is **TRA$B_CLASS**.

**BYTE 3** indicates the type of this record. The field name is **TRA$B_TYPE**.

**BYTE 4** indicates the version of the file format. This version is incremented by DECSIM developers when they have made changes in the file format that would cause a correctly written applications program to fail. The field name is **TRA$B_MAJVER**.

**BYTE 5** indicates the minor version of the file format. This number is incremented by DECSIM developers when they have made changes in the file format that would not cause a correctly written applications program to fail. An example of a minor change is the addition of a new type of record or a new field. The field name is **TRA$B_MINVER**.

**BYTE 6** indicates the revision number of the file format. This number is for bug-tracking purposes and is relevant only to DECSIM developers. The field name is **TRA$B_REVISION**.

**BYTES 7–14** contain the time, in VAX/VMS absolute date and time format, when the module was created. The field name is **TRA$R_CRETIM**.

**BYTES 15–22** can be updated by the applications program to indicate the last time the module was written to by any process other than the original creating process. The field name is **TRA$R_MODTIM**.

**BYTES 23–end of record** hold **TRA$T_MODNAM**, which contains a counted string in its two subfields, **TRA$W_MODNAMLLEN** and **TRA$T_MODNAMSTR**. **TRA$W_MODNAMLLEN** is 2 bytes in length and indicates the module name length in bytes. **TRA$T_MODNAMSTR** contains the module name itself.
G.2.1.2 Module Description Record (MDR)

The Module Description Record (Class 1, Type 2) is the second record in a binary trace file and contains detailed information describing the contents of the binary trace module. Only one Module Description Record is present in a binary trace module. This record is followed by the Node Name Data Set.

The following example, generated from a binary trace module by the ANALYZE/RMS utility, illustrates the structure of the MDR record.

```
DATA BYTES (VBN 1, offset %'0020'):

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>1A</td>
<td>01</td>
<td>02</td>
<td>01</td>
<td>00</td>
<td>26</td>
</tr>
<tr>
<td>04</td>
<td>04</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
```

BYTES 0–3 indicate the record length, class, and type, as described above in the Main Module Header record.

BYTE 4 indicates the type of description record. The field name is TRAS$B_SUBTYPE.

BYTE 5 is a bit vector with Boolean flags that indicate the type of trace. The trace types and field names that are associated with each bit flag are as follows:

- BIT 0: TRAS$V_VECTOR: vector trace
- BIT 1: TRAS$V_NORMAL: normal trace
- BIT 2: TRAS$V_DELAYED: delayed trace
- BIT 3: TRAS$V_IMMEDIATE: immediate trace
- BIT 4: TRAS$V_FORCED: forced trace

Note that more than one flag may be turned on. In the example above, the contents of this field, 1A, translate to 00011010 in binary, indicating that this is a normal, immediate, forced trace. The field name is TRAS$B_TRATYP.

BYTES 6–13 are an obsolete field, TRAS$Q_BASETIME, which used to contain the simulation time when the trace was created. In binary files of Version 2.0–1 and greater, this value is added to appropriate time fields in the Node State Data Set. The field name is now TRAS$Q_OBSOLETE_0.

BYTE 14 indicates what time units apply to the simulation time values reported in the Node State Data Set. The numbers and field names associated with the valid time units are as follows:

- 6: TRAS$F_SEC: femtoseconds
- 5: TRAS$P_SEC: picoseconds
- 4: TRAS$N_SEC: nanoseconds
- 3: TRAS$U_SEC: microseconds
- 2: TRAS$M_SEC: milliseconds
Binary Trace File Format

- 1: TRA$SEC: seconds

The field name is TRA$B_SIM_UNITS. In the example above, the simulation time units are nanoseconds.

BYTE 15 indicates the units applied to time values when reported to the user. The numbers and field names associated with the valid time units are as described above. The field name is TRA$B_USER_UNITS.

BYTES 16–17 is the value of the multiplier applied to the TRA$B_SIM_UNITS. If the simulation resolution is 10 picoseconds, then TRA$B_SIM_UNITS will be TRA$PSEC, while TRA$B_MULTIPLIER will have a value of 10. The field name is TRA$W_MULTIPLIER.

BYTES 18–36 are reserved and should always be zero. The field name is TRA$R_RESERVED.

BYTE 37 indicates the type of fault that was placed on the faulted node. The numbers and field names associated with the DECSIM fault types are:

- 1: TRA$K_UNFAULTED: no fault
- 2: TRA$K_STUONE: stuck at 1
- 3: TRA$K_STUZER: stuck at 0
- 4: TRA$K_INPUTONE: input stuck at 1
- 5: TRA$K_INPUTZER: input stuck at 0
- 6: TRA$K_OUTPUTONE: output stuck at 1
- 7: TRA$K_OUTPUTZER: output stuck at 0

The field name is TRA$B_FAULTYP. In the example above, the field contents indicate that no faults have been set.

BYTES 38–end of record hold TRA$T_FAULTAM, which contains a counted string in its two subfields, TRA$W_FAULTAML and TRA$TFAULTAMSTR. TRA$W_FAULTAML is 2 bytes long and indicates the length of the faulted node name in bytes. TRA$T_FAULTAMSTR contains the faulted node name itself. The name is an empty string if no node is faulted.

The fault type and fault name information supply the context information for the contents of the module. If the fault type is any type other than TRA$K_UNFAULTED, the fault name is the name of the node at which the fault is present. The state information contained in the Node State Data Set is then the states that appear in the network with that fault present on the named node.
G.2.2 Node Name Data Set

The Node Name Data Set contains information about the names of the nodes involved in the trace. The data set contains the following types of records:

- A single Node Name Section record (NNS)
- Node Format Data records (NFD)
- Node Name Records (NMR)
- Data Comment Records (DCR)

The Node Name Records and the Data Comment Records may not appear in some trace modules.

G.2.2.1 Node Name Section (NNS)

The Node Name Section record is a Class 2 (Section Identifier Record), Type 1 record that introduces the records containing the names of all nodes whose states are recorded in the binary trace module. Only one of these sections may appear in a module, and this section must appear before the Node State Data Set.

The following example, generated from a binary trace module by the ANALYZE/RMS utility, illustrates the structure of the NNS record.

```
DATA BYTES (VBN 1, offset %X'0048'):
  7 6 5 4 3 2 1 0
  01234567
  01 02 00 02| 0000 |.... |
```

BYTES 0–3 contain the record length, class, and type, as described in the Main Module Header record above.

G.2.2.2 Node Format Data (NFD)

The Node Format Data record Class 3 (Data Record), Type 2 describes the type, size, and position of data within a data buffer and data set. This record or records must be followed immediately by a Node Name record (or a Data Comment Record).

The following example, generated from a binary trace module by the ANALYZE/RMS utility, illustrates the structure of the NFD record.

```
DATA BYTES (VBN 1, offset %X'004C'):
  7 6 5 4 3 2 1 0
  01234567
  00 00 01 02 03 00 13| 0000 |.........|
  02 00 00 00 00 02 E8 00| 0008 |.......
  00 00 00 00 60 00| 0010 |....|
```

BYTES 0–3 indicate the record length, class, and type, as described above in the Main Module Header record.

BYTES 4 indicates whether the data in the Node State Data Set is two-state (1) or four-state (2). The field name is TRAB_DATTYP.
BYTES 5–8 hold an arbitrary 32-bit number that uniquely identifies a node. The field name is **TRA$1_NODNUM**.

BYTES 9–12 is a DECSIM-specific value, the offset of the node into fixed topology. The field name is **TRA$1_NODOFF**.

BYTES 13–14 designate which record in a data set the record position and length information refer to. This number increments only when the maximum size of an RMS record (32,763 bytes) is exceeded. The field name is **TRA$W_RECNUM_A**.

BYTES 15–16 is the length in bits of the field in the Node State Data Set that stores the state information. The field name is **TRA$W_BITLEN**.

States are represented with the minimum number of bits possible. Thus 2-state data is represented with only a single bit (0 = zero, 1 = one), while 4-state data is represented with two bits (00 = zero, 01 = one, 10 = Z, 11 = U). The vector type data is represented in the same manner, where the lowest order data appears at the lowest address and bit offset.

BYTES 17–20 is the bit offset from the beginning of the Node State Record (described in the Node State Data Set below) to the beginning of the data. The field name is **TRA$1_BITPOS**.

---

### G.2.2.3 Node Name Record (NNR)

The Node Name Record (Class 3, Type 3) contains the name of the node whose data and type are described by the immediately preceding Node Format Data record.

The following example, generated from a binary trace module by the ANALYZE/RMS utility, illustrates the structure of the NNR record.

```
DATA BYTES (VBN 1, offset %'0062'):
  7 6 5 4 3 2 1 0
    01234567
    ------------
       --------
  31 41 00 02 03 03 00 06| 0000 |......A1|
```

BYTES 0–3 indicate the record length, class, and type, as described above in the Main Module Header record.

BYTES 4–end of record hold **TRA$T_NODNAM**, which contains a counted string in its two subfields, **TRA$W_NODNAMLEN** and **TRA$T_NODNAMSTR**. **TRA$W_NODNAMLLEN** is 2 bytes in length and indicates the length of the node name in bytes. **TRA$T_NODNAMSTR** contains the node name itself.
Binary Trace File Format

G.2.2.4 Data Comment Record (DCR)
The Data Comment Records (Class 3, Type 1) are optional records which may appear anywhere a Class 3 Record is valid, including between records of a data set.

BYTES 0–3 indicate the record length, class, and type, as described above in the Main Module Header record.

BYTES 4–end of record hold TRAST_DATCOM, which contains a counted string in its two subfields, TRASTW_DATCOMLEN and TRAST_DATCOMSTR. TRASTW_DATCOMLEN is 2 bytes in length and indicates the length of the data comment in bytes. TRAST_DATCOMSTR contains the data comment itself.

G.2.3 Node State Data Set

The Node State Data Set contains the node transitions that occurred during the trace. This data set consists of a Node State Section record and either Node State Records or Node Transition Records, depending on whether the trace is vector or normal. The set may also include the Data Comment Records described above in the Node Name Data Set.

If the trace is a vector trace, the data set contains Node State Records (NSR). An NSR is created during a vector trace when a node transitions. It contains the state of all nodes at that simulation time, whether or not they have transitioned.

If the trace is a normal trace, the data set contains Node Transition Records (NTR). An NTR is created during a normal trace when a node transitions. It contains the state of all the nodes that transitioned at that simulation time. It does not contain the states of the nodes that did not transition.

G.2.3.1 Node State Section (NSS)
The Node State Section record (Class 2, Type 2) introduces the records containing the node state data. The record types that may appear after a Node State Section are:

• Data Comment Records
• Node State Records
• Node Transition Records

The following example, generated from a binary trace module by the ANALYZE/RMS utility, illustrates the structure of the NSS record.

DATA BYTES (VBN 1, offset '0140'):

```
7 6 5 4 3 2 1 0

------------------------
01234567

02 02 00 02 | 0000 | ... | 1
```

BYTES 0–3 contain the record length, class, and type, as described in the Main Module Header record above.
G.2.3.2 Node State Records (NSR)
The Node State Records (Class 3, Type 4) are variable length records containing the node state information. NSRs are generated during a vector trace whenever an individual node transitions, unless the /EVERY, /CALL, or /RETURN qualifiers are used to trigger the trace. NSRs contain the states of all node involved in the trace, regardless of whether or not they transitioned.

The following example, generated from a binary trace module by the ANALYZE/RMS utility, illustrates the structure of the NSR record.

```
DATA BYTES (VBN 1, offset %’0082’):
    7 6 5 4 3 2 1 0
---------------------
00 C8 00 00 04 03 00 00 | 0000 |......E.| 00 55 00 00 00 00 00 00 | 0008 |......U.|
```

BYTES 0–3 contain the record length, class, and type, as described in the Main Module Header record above.

BYTES 4–5 hold the number of the record within a data set. The first record in a data set has a sequence number of zero; the remaining numbers increment by one until the last record in the data set is found. (For n records, n – 1 will be the highest sequence number). The field name is TRA$W_RECNUM.

BYTES 6–13 contains the simulation time when the record is produced, in simulator units. The TRA$T_TIME_LO and the TRA$T_TIME_HI fields map onto the low and high order words, respectively, of TRA$Q_TIME.

The time value stored in this field is in DECSIM's internal time units. You need to divide this value by 2 to see the time you expect.

BYTES 14–end of record contain the state of all the nodes in the trace at the simulation time when the NSR was generated. The field name is TRA$R_STABUF.

States are represented with the minimum number of bits possible. Thus 2-state data is represented with only a single bit (0 = zero, 1 = one), while 4-state data is represented with two bits (00 = zero, 01 = one, 10 = Z, 11 = U). The vector type data is represented in the same manner, where the lowest order data appears at the lowest address and bit offset.

In the example above, the node being traced is a 4-bit vector. The contents of the TRA$R_STABUF field, 55, translate to 01010101 in binary, indicating that the state of the vector is 1111.
Node Transition Records (NTR)

Node Transitions Records (Class 3, Type 5) are generated during a normal trace whenever a node transitions. If other nodes transition at the same time, they will be included in the same NTR. Unlike NSRs, NTRs do not contain the states of the nodes that did not transition.

A Node Transition Record contains a data packet for each node whose state changed. Each packet consists of the 4-byte node number assigned to the node in the NFD record, followed by the bits of the new state. The packets are packed without any separating or alignment bits.

States are represented with the minimum number of bits possible. Thus 2-state data is represented with a single bit (0 = zero, 1 = one), while 4-state data is represented with two bits (00 = zero, 01 = one, 10 = Z, 11 = U). The vector type data is represented in the same manner, where the lowest-order data appears at the lowest address and bit offset. The number of bits used to hold the state data is described by the length field of the corresponding Node Format Data record.

The following example, generated from a binary trace module by the ANALYZE/RMS utility, illustrates the structure of the NTR record.

```
DATA BYTES (VBN 1, offset %X'0144'):

    7 6 5 4 3 2 1 0 01234567
------------------------  ------
19 3C 00 03 05 03 00 19| 0000 |........<.|
00 08 00 00 00 00 00 00| 0008 |.........|
00 70 00 00 00 18 00 00| 0010 |.......p.|
00 00 00 00 00 00 00 00| 0018 |........|
```

BYTES 0–3 contain the record length, class, and type, as described in the Main Module Header record above.

BYTES 4–5 is the number of data packets present in the buffer. The field name is TRAWSW_DATPACCOU.

BYTES 6–13 contain the simulation time of the transitions, in simulator units. The TRAWSL_TRATIM_LO and TRAWSL_TRATIM_HI fields map onto the low and high order words, respectively, of TRAWSQ_TRATIM.

The time value stored in this field is in DECSIM's internal time units. You need to divide this value by 2 to see the time you expect.

BYTES 14–end of records is the set of data packets describing the new states of nodes which have transitioned at the current time. The field name is TRAWSR_TRABUF.

In the example above, there are 3 data packets, arranged as follows.

BYTES 14–17 contain the node number of the first node that transitioned, node 08. BYTE 18<1:0> contains the new state of the node, 0.

BYTES 18<7:2>--22<1:0> contain the node number of the second node that transitioned, node 06. BYTE 22<3:2> contains the new state of the node, 0.

BYTES 22<7:4>--25<3:0> contain the node number of the third node that transitioned, node 07. BYTE 25<5:4> contains the new state of the node, 0.
G.2.4 Module Trailer Records (MTR)

The Module Trailer Records (Class 4) signify the end of the module and contain status information regarding the termination state of the module. These records are of the End Module Status type.

G.2.4.1 End Module Status (EMS)

The End Module Status Record (Class 4, Type 1) must be the last record in the module. It may be followed only by the end of the file or by a Main Module Header record. The status longword is a VAX/VMS condition code indicating the circumstances under which the module was terminated. (See the Introduction to VAX/VMS System Routines for the interpretation of severity codes.) It is an error if no End Module Status record is present.

The following example, generated from a binary trace module by the ANALYZE/RMS utility, illustrates the structure of the EMS record.

```
DATA BYTES (VBN 5, offset %X'000C'):
    7 6 5 4 3 2 1 0                   01234567
----------       ----------
00 00 00 01 01 04 00 06| 0000 | ........|
```

BYTES 0–3 contain the record length, class, and type, as described in the Main Module Header record above.

BYTES 4–7 contain the VAX/VMS condition code indicating the status of the module.

G.3 Extracting Trace Data from Binary Trace Files

To interpret the Node State Data Set correctly, your applications program must use the information in the Node Name Data set to extract the name, simulation time, and state information from the NTRs or NSRs. In some cases, it is necessary to convert that information into the form that the user expects.

A sample program, 'bintradmp', which dumps out the binary records in sequential order in ASCII format, is included in the DECSIM kit.

G.3.1 Extracting Simulation Times from NSRs and NTRs

To extract the simulation times from the Node State Records or Node Transition Records and convert them to the format the user expects, do the following:

1. Concatenate the low- and high-order words in the time fields into a single quadword, dropping the least significant bit and the most significant bit (which are always 0) from the low_order word. See Figure G–3 below.

2. Multiply this value by the value in the TRA$W_MULTIPLIER field in the MDR record.
3 If the values in the TRA$B$SIM$\_UNITS$ and the TRA$B$USER$\_\_\_UNITS$ fields in the MDR record are not the same, apply the appropriate scaling factor.

4 Divide the result by 2. (This is because the binary file time fields contain DECSIM's internal time representations, which are twice the values that the user expects.)

5 Apply the time units specified in TRA$B$USER$\_\_\_UNITS$.

**Figure G–3 Time Field Concatenation**

![Diagram of time field concatenation with high-order and low-order words]

---

**G.3.2 Extracting State Information from NSRs and NTRs**

The methods of extracting the state information from NSRs and NTRs are slightly different.

With NSRs, which store the state data for all nodes involved in the trace, use the bit position to determine the beginning of the state data for each node, and the bit length to determine the end of the data. The bit position and the bit length for each node are stored in the appropriate NFID record.

In NTRs, the state data bits begin immediately after the node number, with no separating or aligning bits. Use the bit length to determine the end of the data bits.

The minimum number of bits is used to store the state. If the data is 2-state, a single bit is used to store either a 0 or a 1. If the data is 4-state, two bits are used. To convert the 4-state data to the form the user expects, change the values as follows:

- 00 = 0
- 01 = 1
- 10 = Z
- 11 = U
G.3.3 Extracting Node Names from NSRs and NTRs

These records do not contain the nodes names that the user expects to see; those names are stored in the Node Name Records.

In NSRs, which hold the state information for all nodes involved in the trace, you simply print out the node names in order as they appear in the NNRs with the corresponding state information from the NSR. Use the bit position to determine which state information is associated with which node.

NTRs store the unique node numbers that are assigned in the NFD records. Use that number to pull out the corresponding NNR which stores the node name used by DECSIM.
Index

- binary minus operator • D–10
- line continuations • 4–5
- unary minus operator • D–8

| comments |
| in behavior models • 2–1 |
| in command lines • 4–4 |

| within quotation marks • 2–3 |
| enclosing names • 2–2, 2–5 |
| in model parameter lists • 3–6 |
| in names • 1–6, 4–9 |

| # |
| radix operator • 4–11, 4–14 |

| $ |
| DCL prompt • 5–236 |
| in structural models • 3–54 |
| text concatenation in behavior macros • 2–20 |

| $$ |
| text concatenation operator for command language macros • 5–146 |

| & |
| concatenation operator • D–21 |

| . |
| in text strings • 2–1 |

| . . |
| enclosing spaces inside double quotes • 2–5 |
| in behavior macros • 2–20 |
| in model names • 3–5 |
| in model parameter lists • 3–6 |
| in strings • 4–9 |
| marking text strings • 2–3 |

| . |
| multiplication operator • D–12 |
| wildcard character • 4–23, 4–31 |

| + |
| binary plus operator • D–10 |

| : |
| in behavior models • 2–5 |

| / |
| division operator • D–12 |

| : |
| in labels • 2–6 |

( )
parentheses operator • D–30
;
in behavior macros • 2–19
in behavior models • 2–5
< >
bit subscript operator • D–14
in behavior models • 2–4
=
assignment operator • D–6
?
in command language expressions • D–5
in PATTERN, DEPOSIT, or SELECT • 4–12
wildcard character • 4–23
@
INDIRECT command • 5–112
[]
in behavior models • 2–4
{}
behavior directives • 2–59
timing directive • 2–52
^
in text strings • 2–1

A

Abbreviations

| in the command language • 4–3 |
| of command language state names • 4–21 |
| of command macro names • 5–145 |
| of symbol names • 4–4, 4–21 |

Absolute time • 5–210

| as a command argument • 4–13 |
| in the DEPOSIT command • 5–32 |

Accessing model from the command language • 4–20
Accessing unnamed signals in a model • 4–21

$ACCESSLIST macro template parameter • 5–144
Access lists

| in DECSIM commands • 4–21 |
| ACTIVATE command • 5–2 |
| ACTIVATE statement • 2–47 |

| and routines that return a value • 2–46 |
| compared to routine call • 2–47 |

Actual parameters

| and formal parameters • 3–48 |

Index–1
Index

Actual parameters (cont'd.)
in an instance statement 3-12, 3-48
in behavior macros 2-21
in behavior routines 2-25
passing of 3-47 to 3-52
AND operator 2-18
Area units
in structural models 3-56
Arithmetic representation
in behavior models 2-63
Array declarations
in behavior states 2-14
%ARRAY ELEMENT 5-223, 5-242
ASCII characters
use in DECSIM languages B-1
Assertion/polarity signal names 4-26
Assigning values
to behavior ports 2-12
to behavior states 2-16
from the command language 2-16
to bidirectional ports 2-29
AXE
driving a DECSIM model with 5-184

Behavior directives
operator directives (cont'd.)
%CARRY 2-61
%CARRYIN 2-61
%OVERFLOW 2-62
TC 2-63
US 2-63
WIDTH 2-60
scope of 2-59
time directives 2-62
TIMESCALE 2-60
Behavior language
character set in 2-1, B-1
comments in 2-1
connectives in 2-5
deimiters in 2-5
labels C-4
names C-4
numbers and radix in 2-6
subscripts in 2-4 to 2-5
synonyms C-4
text strings in 2-3
Behavior models
and behavior modules 2-9
assigning values to ports 2-12
assigning values to states 2-16
communication between models 2-15
compiling 2-11, 2-19, 5-15
declarations in 2-11
definition header 2-8
definition of 2-7
keywords 2-8
names C-4
noninterconnected 2-8
reading output ports 2-13
subblock notes C-4
time in 2-65
twostate default in 2-30
Behavior modifiers
BIDIRECTIONAL 2-29
CONVxx 2-32
definition of 2-28
ENTRY 2-30
FOURSTATE 2-30
INCLUDED 2-36
INITIAL 2-35
INPUT 2-10, 2-28
NOENTRY 2-31
OUTPUT 2-10, 2-28
STATIC 2-35
TWOSTATE 2-29
Behavior modifiers (cont'd.)
    UNCONNECTED_OK • 2–34
Behavior modules
    and behavior models • 2–9
    compiling • 2–11, 5–15
    declarations in • 2–9
    definition • 2–8
    modifiers • 2–10
    module header • 2–9
    names of • 2–9, C–4
Behavior ports
    assigning values to • 2–12
    initialization • 2–34
Behavior primitives
    definition of • 2–1
Behavior routines
    accessing formal parameters in • 4–29
    accessing from the command language • 4–28
    accessing return values • 2–46
    activating with the ENTRY modifier • 2–30
    calling • 2–53
    non-BDS routines • 2–54
    with the CALL command • 5–13
    calling and activating • 2–47
    communication between routines • 2–15
    declaring external routines • 2–27
    declaring non-BDS routines • 2–27
    defining in a model • 2–23
    entry routine • 2–31
    header • 2–23
    Initializing • 2–26
    names • 2–23, C–4
    restrictions • 2–24
    return value • 2–24, 5–14
    scheduling with the ACTIVATE command • 2–47, 5–2
    tasks • 2–47
    triggering with assignments • 2–52
    using routine calls as expressions • 2–54
Behavior statements (cont'd.)
    REPEAT • 2–41
    RETURN • 2–45
    SELECT • 2–42
    SELECTALL • 2–44
    SELECTONE • 2–43
    UNTIL-DO • 2–40
    WAIT • 2–51
    and local states • 2–51
    WHILE-DO • 2–39
Behavior states
    accessing from the command language • 4–28
    and WATCH, TRACE • 2–15
    assigning values to • 2–16
    displaying value of • 5–66
    global • 4–28
    global and local • 2–15
    initializing • 2–35, 5–127
    local • 4–28
    modifiers in • 2–28
    names • 2–2, C–4
    static and dynamic • 2–15
    BIDIRECTIONAL modifier • 2–29
Bidirectional ports
    assigning values to • 2–29
    BIDIRECTIONAL signal declaration • 3–7, 3–26
Binary minus operator • D–10
Binary plus operator • D–10
Binary trace files
    concatenating • G–2
    creating • 5–215, G–1
    definition of • G–1
    extracting data from • G–13
    in simulation replay • 5–173
    node name data set in • G–8
    node state data set in • G–10
    opening log files for • 5–137, G–2
    reading • G–2
    records
    DCR • G–10
    EMS • G–13
    MDR • G–6
    MHR • G–4
    MMH • G–5
    MTR • G–13
    NFD • G–8
    NNR • G–9
    NNS • G–8
    NSR • G–11
    NSS • G–10
    NTR • G–12
Index

Binary trace files (cont'd.)
   record sequence in • G–3
   specifying new signal names in • 5–218
   using as pattern files • 5–154
Bit mask • 4–15
Bit subscript operator • D–14
Boolean operators
   definition of • D–16
BREAK command • 4–36, 5–7
BUF operator • D–16

C

*C • 4–32, 4–34
CALL command • 5–13
   and HALT statement • 2–41
Calling behavior routines • 2–53
CANCEL ACTIVATE command • 5–2
CANCEL BREAK command • 5–9
CANCEL CONTEXT command • 5–25
CANCEL DETECT command • 5–50
CANCEL ERROR command • 5–61
CANCEL FAULT command • 5–82
CANCEL HAZARD command • 5–102
CANCEL INDIRECT command • 5–114
CANCEL LOG command • 5–139
CANCEL MACRO command • 5–150
CANCEL PATTERN command • 5–153
CANCEL RADIX command • 5–169
CANCEL SAVE command • 5–177
CANCEL SCOPE • 5–179
CANCEL STATE command • 5–200
CANCEL TRACE command • 5–226
CANCEL WATCH command • 5–244
Capacitance
   in the process parameter file • F–7
CAPACITANCE declaration • 3–38
Capacitance units
   in structural models • 3–55
CAPDA parameter
   in the process parameter file • F–7
CAPDP parameter
   in the process parameter file • F–7
CAPGA parameter
   in the process parameter file • F–3
%CARRY
   operator directive in behavior models • 2–61
%CARRYIN
   operator directive in behavior models • 2–61
Case keywords (cont'd.)
   in the process parameter file • F–10
Case selection
   in behavior models • 2–42, 2–43, 2–44
CHAIN_INTER_CONTRIB
   in the process parameter file • F–3
Character set
   in behavior language • B–1
   in behavior models • 2–1
   in command language • B–1
   in network description language • B–1
%CHAR print element in %PRINT statement • 2–50
COMBEH (a command macro) • 5–15
COMBEH/LIS
   example of listing file • 2–22
Command arguments • 4–8 to 4–20
   expressions • 4–14 to 4–20
   file specifications • 4–10
   integer numeric values • 4–13
   logic values • 4–11
   quoted text strings • 4–9
   time values • 4–13
Command blocks • 5–5, 5–110, 5–182
debugging • 5–31
exiting from • 5–125
Command format • 5–1
Command language conventions • 4–2 to 4–8
   abbreviations • 4–3
   character set • B–1
   command format • 4–2
   comments in the command line • 4–4
   error handling • 4–6 to 4–8
   labels • 4–5
   line continuations • 4–5
Command language states
   accessing • 4–29
   as operands in expressions • 4–20
   creating • 5–200
   initializing • 5–127
   naming of • 4–21, C–3
Command modes • 4–32
Command prefixes • A–1
Command prompts • 4–32
CTRL–C • 4–32
MACRO • 4–32
MORE • 4–32
PATETERNAME • 4–33
QUOTE • 4–32
SIM • 4–32
SMBREAK • 4–33
XXX • 4–32
Command prompts (cont'd.)  
@XXX> • 4-32
Commas  
in behavior models • 2-5
Comments  
in behavior models • 2-1  
in command lines • 4-4  
passing to terminal • 5-118
COMPILE/PROCESS command • 3-2
COMPILE command • 5-18
Compiler lexical statements  
In structural models • 3-52
Compiling  
behavior models • 2-19  
behavior modules, models • 2-11  
MOS models • 5-20  
structural models • 5-18
Concatenation operator • D-21  
for command language macros • 5-146  
in behavior macros • 2-20
Conditional command execution • 5-110, 5-182
Conditional hazard checkers  
See also Hazard checkers  
defined in a model • 1-22  
defined in the command language • 5-100  
violations and boundary conditions • 4-54
CONFIGURE command • 5-23
Connection lists  
in instance statements • 3-14 to 3-17  
in memory primitives • 1-15, 1-16  
in MOSFET statements • 1-10  
in RESISTOR statements • 1-11  
in structural models • 3-6  
  bidirectional • 3-7
CONSTANT declaration  
in behavior models • 2-17
Constants  
as operands in equation statements • 1-6  
as operands in expressions • 4-14  
extracting bits from behavioral • 2-59  
in behavior models • 2-17  
width of • 4-14
CONTEXT command • 5-25
CONTINUE (a command macro) • 5-27
Control characters  
in %PRINT statement • 2-49
Control flow commands • 5-5, 5-88, 5-110, 5-125, 5-182, 5-248
Control flow tracing • 5-31
CONVOZ operator • 3-25
CONV1Z operator • 3-25
Conversion operators  
CONV4 • D-24  
definition of • D-24  
fourstate data conversion • D-25
CONVxx modifier • 2-32
%CRLF print element in %PRINT statement • 2-50
CTRL/C • 4-34
  and EXIT, QUIT • 5-73
CTRL/G • 4-7, 4-34
Cyclewidth checkers  
See also Hazard checkers  
defined in a model • 1-21  
defined in the command language • 5-99  
vviolations and boundary conditions • 4-45

D

DEPOSIT command • 4-3
Data format representations • D-26
%DATE print element in %PRINT statement • 2-50  
%DAYTIME print element in %PRINT statement • 2-50
DCL command • 5-29
DEBUG command • 5-30
Debug mode • 5-7
Declarations  
in behavior models • 2-11
DECSIM command qualifiers  
/CPUSTIME • 4-1  
/INPUT • 4-1  
/[NO]INITIALIZE • 4-1  
/OUTPUT • 4-1
DECSIM data structures  
accessing address of behavior data • 2-54  
accessing behavior data • 2-55  
accessing from external program • 5-184  
fields in • 5-188  
fourstate or wide data • 5-188  
global section layout • 5-186
DECSIM initialization • 4-1
DECSIM.INI • 4-1
DECSIM.SYS • 4-1
DEFAULT_TIMEOUT parameter  
in the process parameter file • F-3
DEFAULT_TIMEOUT statement • 3-46
DELAY declaration • 3-41
Delayed assignment  
direct assignment • 2-52

Index -5
Delayed assignment (cont'd.)
and routine invocation • 2–30
in behavior models • 2–52
restriction on • 2–53
timing directive • 2–52
to behavior arrays • 2–15
to ports • 2–12
to states • 2–16
DELTA-L keyword • 1–12
DELTA-W keyword • 1–12
DELTA_L keyword
in the process parameter file • F–7
DELTA_W keyword
in the process parameter file • F–7
DEPOSIT/FORCE command
with behavior vectors • 3–34
DEPOSIT command • 5–32
DETECT command • 5–36
DEVICE SECTION
in the process parameter file • F–9
Direct assignment
in behavior models • 2–52, 2–53
DISABLE BREAK command • 5–11
DISABLE DETECT command • 5–51
DISABLE HAZARD command • 5–102
DISABLE INDIRECT command • 5–115
DISABLE LOG command • 5–142
DISABLE PATTERN command • 5–153
DISABLE TRACE command • 5–226
DISABLE WATCH command • 5–244
Distance units
in structural models • 3–56
Division operator • / • D–12
DO–UNTIL statement • 2–40
DO–WHILE statement • 2–39
DROP command • 5–56

Emergency halts (cont'd.)
interactive • 4–34
ENABLE BREAK command • 5–11
ENABLE DETECT command • 5–51
ENABLE HAZARD command • 5–102
ENABLE INDIRECT command • 5–115
ENABLE LOG command • 5–142
ENABLE PATTERN command • 5–153
ENABLE TRACE command • 5–226
ENABLE WATCH command • 5–244
ENTRY modifier • 2–30
EQL operator • D–31
EQL(QUOTE) operator • D–31
EQN keyword • 1–2
Equation statements • 1–1 to 1–8
See also propagation delay
definition and format of • 1–1
expressions in • 1–5 to 1–7
feedback in • 1–7
inputs in • 1–1, 1–5ff, 1–6
name-based parameters in • 1–2
order-based parameters in • 1–3
outputs in • 1–1, 1–5
restrictions for • 1–7
sample, in a model • 1–1
EQV operator • D–18
%ERASE_SCREEN print element in %PRINT
statement • 2–50
ERROR command • 5–61
Error handling
of errors in blocks and loops • 4–8
of errors in command language semantics • 4–7
of errors in command language syntax • 4–6
of errors in EVALUATE statements • 4–7
of errors in expressions • 4–7
of errors in indirect files • 4–8
Error messages
specifying handling of • 5–61
types of in DECSIM • 4–6
Errors
 correcting interactive errors • 4–7, 4–34
Escape prompting • 4–33
restrictions • 4–34
EVALUATE command • 5–64
Event-driven simulation • 2–65
EXAMINE command • 5–66
EXE files • 5–128
EXIT command • 5–73
EXPLICIT_NODE declaration • 3–44
$EXPRESSION macro template parameter • 5–144
Expression operands

Index—6
Expression operands (cont'd.)
  command language states • 4–20
  constants • 4–14
  in behavior models • 2–57
  in command arguments • 4–14
  in equation statements • 1–5
  subexpressions • 4–20

Expression operators
  and behavior language directives • D–3
  and the wildcard ? • D–5
  and two-state, four-state data • D–3
  arithmetic
    binary minus • D–10
    binary plus • D–10
    division / • D–12
    MOD / D–12
    multiplication • D–12
    unary minus • D–8
  assignment = • D–6
  bit subscript <> • D–14

Boolean
  AND • D–18
  BUF • D–16
  EQV • D–18
  NAND • D–18
  NOR • D–18
  NOT • D–17
  OR • D–18
  XOR • D–18
  concatenation & • D–21
  conversion CONVxx • D–24, D–25
  extension
    OXT • D–26
    SXT • D–26
    ZXT • D–26
  in behavior models • 2–57
  in command arguments • 4–14, 4–20
  in equation statements • 1–6
  MAP • D–28
  parentheses () • D–30
  relational
    EQL • D–31
    EQL(QUOTE) • D–31
    GTR • D–31
    LEQ • D–31
    LSS • D–31
    NEQ • D–31
    NEQ(QUOTE) • D–31
  relative precedence table • D–2
  shift operators SLx, SRx • D–33
  tristate
    MOSTS • D–35

Expression operators (cont'd.)
  tristate (cont'd.)
    TS • D–35
    TSCON • D–35

Expressions
  as command arguments • 4–14 to 4–20
  as time values in behavior language • 2–62
  as time values in command language • 4–13
  defaults • D–5
  error handling of • 4–7
  evaluation of • 5–64
  in behavior models • 2–56 to 2–59
    field extraction • 2–59
    order of evaluation • 2–56
    width of result • 2–56
  in delayed assignments • 2–52, 2–62
  in equation statements • 1–5 to 1–7
  in subscripts • 2–4, 2–5
  nesting of • 4–20
    in behavior models • 2–59
  routine calls • 2–54
  syntax of • 4–14

Expressions in %PRINT statement • 2–51
Extension operators • D–26
EXTERNAL ROUTINE declaration
  in behavior models • 2–27

F

Fault collapsing • 4–36
FAULT command • 5–75
Fault detection
  disabling and enabling fault detectors • 5–51
  removing fault detectors • 5–50
  setting fault detectors • 5–36
  showing current status • 5–52
Fault effect • 5–44
Fauxts
  displaying • 5–84
  dropping from the network • 5–56
  inserting in a network • 5–75
  removing from a network • 5–82
Fault simulation • 4–36
  and machine descriptors • 4–30
  using wildcards in • 4–31
Fault sourcing • 4–36
FE (ZYCAD Fault Evaluator) • 4–37
Feedback
  in equation statements • 1–7
Field extraction

Index–7
Index

Field extraction (cont’d.)
   from constants and expressions • 2–59
   using the bit subscript operator • D–15

File extensions
   .EXE • 5–128
   .MEM • 5–130
   .NET • 3–1
   .NLB • 3–1
   .NOB • 3–2
   .NRQ • 3–52
   .NSV • 5–177
   .ULD • 5–130
   .ZBR • 5–192, 5–250
   .ZOF • 5–192

$FILE macro template parameter • 5–144

Files
   displaying on the terminal • 5–229
   editing within DECSIM • 5–57

File specifications
   as command arguments • 4–10
   defaults • 4–10
   special characters in • 4–10

FOR command • 5–88

Foreign language routines
   calling • 2–54
   INCLUDED modifier • 2–36
   writing • 2–56

FOREVER • 2–51

Formal parameters
   and actual parameters • 3–48
   in a model header • 3–5
   in behavior macros • 2–19
   in behavior routines • 2–24

FOR statement • 2–38

FORWARD ROUTINE declaration
   in behavior models • 2–26

Fourstate data conversions • D–25, D–28

FOURSTATE modifier • 2–30

Fourstate variables
   and the tristate operator • D–35
   restriction on in FOR loop • 2–38
   restriction on in SELECT • 2–42
   restriction on in SELECTALL • 2–44
   restriction on in SELECTONE • 2–43
   restriction on in UNTIL-DO • 2–40
   restriction on in WHILE-DO • 2–39

Global declarations
   in behavior models • 2–11
   in structural models • 3–8

GLOBAL subblocks
   in structural models • 3–8

GTR operator • D–31

H

HELP command • 4–3
HALT command • 5–90
HALT statement • 2–41

Hard DECSIM

See ZYCAD

Hazard analysis • 4–39
   features • 4–40
   in the design process • 4–40
   restrictions • 4–40

Hazard checkers
   canceling • 5–102
   capturing output in a log • 1–23
   defining in a model • 1–18 to 1–23
      format • 1–19
      parameter fields in • 1–20
      report format • 1–23
      sample • 1–18
   defining in the command language • 5–91 to 5–101
   disabling and enabling • 5–102
   displaying • 5–105
   restrictions on defining in a model • 1–19
   restrictions on simulation with • 1–23
   violations and boundary conditions • 4–42

HAZARD command • 5–91

Headers
   in behavior models • 2–8
   in behavior routines • 2–23

Help
   escape prompting • 4–33

HELP command • 5–108

Hierarchy
   accessing symbols in • 4–22
   and SET SCOPE • 4–23, 5–179
   %NET label • 4–23
   signal names in • 4–27
   wildcards in hierarchical names • 4–24

Hierarchy in model definitions

See Structural models, nesting of

G

*G • 4–7, 4–34

Index–8
HIGHTHRESH parameter
   in the process parameter file • F–2
Hold checkers
   See also Hazard checkers
   defined in a model • 1–22
   defined in the command language • 5–99
   violations and boundary conditions • 4–51
%HOMEP print element in %PRINT statement • 2–50

I
IC label • 3–10
IDENT parameter
   in the process parameter file • F–2
IF command • 5–110
IF–THEN statement • 2–37
INCLUDE declaration
   in behavior models • 2–27
INCLUDED modifier • 2–36
INDIRECT command • 5–112
Indirect files
   creating from log files • 5–135
   disabling and enabling • 5–115
   displaying • 5–117
   executing • 5–112
   halting execution of • 5–90
   resuming execution of • 5–27, 5–176
   suspending execution • 5–163
   terminating execution of • 5–114
INFORM (a command macro) • 5–118
INFORM declaration
   in behavior models • 2–19
INFORM statement • 3–53
Initialization
   from executable files • 5–128
   from MICRO2 files • 5–130
   from text files • 5–130
   of behavior ports • 2–34
   of behavior states • 2–35
   of input signals in structural models • 3–28, 3–29
   of states and arrays • 5–127
INITIALIZE ROUTINE declaration
   in behavior models • 2–26
INITIAL modifier • 2–35
INPUT modifier • 2–10
INPUT signal declaration • 3–29
Instance statements (cont’d.)
   connection lists in • 3–14 to 3–17
   extracting signals in • 3–19
   format of • 3–10
   labels for • 3–10, 4–25
   model name in • 3–11
   parameters in • 3–48
   specifying subblocks • 3–11
   unconnected signals in • 3–15
   vector signals in • 3–17
Integer values
   as command arguments • 4–13
   in behavior, network, and command languages • C–3
Interactive debugging
   initiating • 5–163
   terminating • 5–176, 5–205
Interactive features • 4–32
Interconnect language
   See Network description language
Interpolating resistance
   in the process parameter file • F–7
Invoking routines
   with the ENTRY modifier • 2–30
IN_STUB signal declaration • 3–27
Iterative command execution • 5–88, 5–248

J
^J • 2–49

K
KEY command • 5–119
Keys
   defining • 5–119
   displaying definitions • 5–122

L
LABEL command
   displaying revision numbers • 2–18
Labels • 4–5
   displaying state of • 5–66
   for behavior statements • 2–6
   for instance statements • 3–10, 4–25, 5–179

Index–9
Index

Labels (cont'd.)
  in behavior, network, and command languages • C–2
  in the behavior language • C–4
  in the command language • 4–5
  printing of • 5–124
LABELS (a command macro) • 5–124
LE(ZYCAD Logic Evaluator)
  architecture of • E–1
LE (ZYCAD Logic Evaluator) • 4–37
LEAVE command • 5–125
LEAVE statement • 2–46
LEQ operator • D–31
Level-sensitive memory
  definition of • 1–13
  specifying in memory primitives • 1–14
Libraries • 1–8
  creating library files • 3–1
  defaults • 3–2
  GATES.NET • 1–1
  search procedure • 3–3, 3–11
  specifying • 3–2, 5–23
LIBRARY statement • 3–2
Line continuations
  in the command language • 4–5
Listing files • 2–22
LOAD command • 5–127
Loading a network • 5–231
Local declarations
  in behavior models • 2–11
%LOC function • 2–54, 2–55
LOG command • 5–134
Log files
  See also Binary trace files
  appending • 5–135
  closing • 5–139
  controlling writing to • 5–142
  deleting • 5–139
  displaying active files • 5–141
  labels for • 5–138, 5–137
  modifying • 5–138
  opening • 5–134
  using as indirect files • 5–135
  writing to with PRINT command • 5–164, 5–166
Logic state values
  and the /RADIX qualifier • 4–12
  and the SET RADIX command • 4–12
  as command arguments • 4–11
  in behavior, network, and command languages • C–3
LOGNAME directive in %PRINT statement • 2–49

Log names
  specifying with SET LOG or /LOGNAME • 4–10
LOWTHRESH parameter
  in the process parameter file • F–2
LSS operator • D–31

M

^M • 2–49
Machine descriptors • 4–30
  and the CONTEXT command • 5–25
  and wildcards • 4–31
MACRO command • 5–143
MACRO declaration
  in behavior models • 2–19
Macros
  command language
    names • C–3
  debugging • 5–30
  defining in the command language • 5–143
  deleting • 5–150
  disabling and enabling • 5–115
  displaying • 5–117, 5–148
  editing • 5–59, 5–60
  in behavior models
    defining • 2–19
    expansion of • 2–22
    names • C–4
    using • 2–21
  in structural models
    concatenation • 3–54
    string manipulation • 3–53
    terminating execution of • 5–114
MAP operator • D–28
MAXRES parameter
  in the process parameter file • F–3
MAXRES statement • 3–47
MEM files • 5–130
Memory arrays
  accessing from the command language • 4–29
  displaying state of • 5–66
  initializing • 5–127
MEMORY keyword • 1–14
Memory primitives • 1–13 to 1–17
  array names in • 4–29
  attributes of • 1–14
  definition and format of • 1–13
  edge-triggered • 1–13
  handling of undefined bits in • 1–17

Index–10
Index

Memory primitives (cont'd.)
  level-sensitive • 1–13
  ports in • 1–15
  READ_PORT parameters in • 1–15
  sample, in a model • 1–13
  WRITE_PORT parameters in • 1–16
MICRO2 files • 5–130
M–module
  in the ZYCAD LE • E–1
Model
  See Behavior model, Structural model
MODEL/ENDMODEL keywords • 3–5
Model library files
  See Libraries
Model names
  in instance statements • 3–11
Modifiers
  See Behavior modifiers
MOD operator • D–12
Module
  See Behavior module
MODULE/ENDMODULE keywords • 2–8
MODULES directive in behavior models • 2–60
MOS command • 5–151
MOSFET keyword • 1–9
MOSFET statements
  See MOS primitives
MOS models
  compiling • 5–20
  signal declarations • 3–37 to 3–45
    CAPACITANCE • 3–38
    DELAY • 3–41
    EXPLICIT_NODE • 3–44
    NO_CHARGE_SHARE • 3–43
    PARASITIC • 3–39
    PCAP • 3–40
    POWER • 3–37
    THRESHOLD • 3–40
    TIMEOUT • 3–43
  specifying a delay time • 5–151
  types of primitives
    in the process parameter file • F–11
MOS nodes
  calculating capacitance of • F–4
  specifying attributes of • 3–37
MOS primitives • 1–9 to 1–12
  definition and format of • 1–9
  MOSFET statements • 1–9 to 1–11
  RESISTOR statements • 1–11 to 1–12
  restrictions for • 1–12
  sample, in a model • 1–9
MOSTS operator • 3–25, D–35
Multibit signals
  concatenating • 3–17
  extracting in an instance statement • 3–19
  in instance statements • 3–17
Multiplication operator • D–12

N

$NAME macro template parameter • 5–144
Names
  for global and local states • 2–2
  in behavior models • 2–2
  in the behavior language • C–4
  labels • C–2
  of behavior modules • C–4
  of command language macros • C–3
  of command language states • C–3
  of model subblocks • C–4
  of parameters • C–2
  of signals • C–2
  of structural or behavior models • C–4
  special characters in C–1
  transistor ID strings • C–4
  user–defined • C–4
NAND operator • D–18
NEQ operator • D–31
NEQ(QUOTE) operator • D–31
NET files • 3–1
%NET label • 4–23, 5–180
Network description files
  format • 3–1
    library • 3–1
    top–level • 3–2
Network description language
  character set in B–1
  definition of • 3–1
%NET_NAME print element in %PRINT statement • 2–50
NLB files • 2–11, 3–1
NOB files • 3–2
NOCRLF directive in %PRINT statement • 2–49
NODE_CAPAC_DEFAULT
  in the process parameter file • F–3
NOENTRY modifier • 2–31
Non–BDS routines • 2–54
Non–event–driven simulation • 2–65
Nonparasitic capacitance
  in the process parameter file • F–3

Index–11
Index

NOR operator • D–18
NOT operator • D–17
NOZEROFILL directive in %PRINT statement • 2–49
NO_CHARGE_SHARE declaration • 3–43
NO_WIRETIE_CHECK signal declaration • 3–36
NRO files • 2–18, 3–52
NSV files • 5–177
Number base
  specifying with RADIX command • 5–169
$NUMBER macro template parameter • 5–144
Numbers
  in behavior models • 2–6

Parameters (cont’d.)
  interpolating missing parameter values • 3–51
Min/Nom/Max • 3–50
  names
    in behavior, network, and command
      languages • C–2
    passing of • 3–47 to 3–52
      example • 3–49
    passing through multiple levels • 3–50
      specifying one value from a triple • 5–212
    that can be passed • 3–49
    time values
      in behavior, network, and command
        languages • C–3
PARAMETER_SECTION
  in the process parameter file • F–2
PARASITIC declaration • 3–39
Parasitic-parameter-type name
  in the process parameter file • F–10
PARASITIC_PARAMETER section
  in the process parameter file • F–5
Parentheses operator • D–30
Passing parameters
  to behavior macros • 2–21
  to behavior routines • 2–24
  PATTERN command • 5–153
Pattern vectors
  applying to network inputs • 5–153
PAUSE (a command macro) • 4–35, 5–163
PCAP declaration • 3–40
Phantoms
  equation operators for • 3–25
  inserting in a network • 3–22
  naming • 3–23
  specifying the type of • 3–21
  specifying time delay in • 3–35
  user-defined • 3–24
Polarity
  conventions for indicating in signal names • 4–26
  in behavior models • 2–2
PORT declarations
  in behavior models • 2–12
  in behavior modules • 2–10
  modifiers in • 2–28
Ports
  in memory primitives • 1–15
  %POSITION print element in %PRINT statement • 2–50
POWER declaration • 3–37
PRECISION statement • 3–45
Primitives

O

OBJ files • 2–27
OLB files • 2–27
Online help • 5–108
  See also Escape prompting
Operands
  See Expression operands
Operators
  See Expression operators
OR operator • D–18
Oscillation
  detecting • 5–36
OUTPUT modifier • 2–10
OUTPUT signal declaration • 3–29
OUT_STUB signal declaration • 3–27
%OVERFLOW
  operator directive in behavior models • 2–62
OXT operator • D–26

P

PAI files
  creating • E–3
  samples • E–4
PAO files • E–3
  samples • E–4
Parameters
  for each primitive type • 3–48
  in delayed assignments • 2–52
  in equation statements • 1–2
  in memory primitives • 1–15, 1–16
  in MOSFET statements • 1–9
  in RESISTOR statements • 1–11

Index–12
Primitives (cont'd.)

See also Behavior primitives, Equation statements, Memory primitives, MOS primitives
definition of • 1-1
interconnecting different types • 3-21 to • 3-25
PRINT command • 5-164
%PRINT statement • 2-48
directives • 2-48
LOGNAME • 2-49
NOCRLF • 2-49
NOZEROFILL • 2-49
RADIX • 2-49
WIDTH • 2-49
ZEROFILL • 2-49
print elements • 2-49
%CHAR • 2-50
%CRLF • 2-50
%DATE • 2-50
%DAYTIME • 2-50
%EREASE_SCREEN • 2-50
expressions • 2-51
%HOMEN • 2-50
%NET_NAME • 2-50
%POSITION • 2-50
%READ_OUTPUT_PORT • 2-50
text strings • 2-50
%TIME • 2-50
Print system variables • 5-165
Process parameter file • 1-12
calculating node capacitance • F-4
DEVICE_SECTION • F-9
case keywords • F-10
parasitic-parameter-type • F-10
primitive MOS models • F-11
resistance-set-names • F-11
syntax • F-9
usage names • F-11
nonparasitic capacitance • F-3
CAPGA • F-3
CHAIN_INTER_CONTRIB • F-3
NODE_CAPAC_DEFAULT • F-3
PARAMETER_SECTION • F-2
DEFAULT_TIMEOUT • F-3
HIGHTRESH • F-2
IDENT • F-2
LOWTHRESH • F-2
MAXRES • F-3
PARASITIC_PARAMETER section • F-5
area • F-5
capacitance • F-7
CAPDA • F-7
Process parameter file
PARASITIC_PARAMETER section (cont'd.)
CAPDP • F-7
coefficients • F-5
DELTA_L • F-7
DELTA_W • F-7
periphery • F-5
RESISTANCE_SECTION • F-7
interpolating resistance • F-7
syntax • F-8
use of in MOS simulation • F-1
Prompts
See Command prompts
Propagation delay
See also Timing delay
NETPRO calculation of • 1-2, 1-3 to • 1-5
specifying in equation statements • 1-2 to • 1-3
specifying in memory primitives • 1-15, 1-16
%PTR function • 2-55
PULLDOWN signal declaration • 3-36
PULLUP signal declaration • 3-36
Pulselength checkers
See also Hazard checkers
defined in a model • 1-21
defined in the command language • 5-99
violations and boundary conditions • 4-42

Q

QUIT command • 4-3
Question mark
as a wildcard • 4-12
QUIT command • 5-73
Quotation marks
enclosing names • 2-5
in model parameter lists • 3-6
in names • 1-6, 2-2, 4-9
within quotation marks • 2-3
$QUOTEDTEXT macro template parameter • 5-144

R

Radix
in behavior models • 2-6
RADIX command • 5-169
RADIX directive IN %PRINT statement • 2-49
/RADIX qualifier
/RADIX qualifier (cont'd.)
effect on display of logic values 4-12
%READ_OUTPUT_PORT 2-13
as print element in %PRINT statement 2-50
RECALL command 5-171
Reference designator 3-10
Relational operators D-31
Relative time 5-210
as a command argument 4-13
in the DEPOSIT command 5-32
Reloading a saved network 5-175
REPEAT statement 2-41
Replayable trace files
creating 5-219
REPLAY command 5-173
Reporting bugs 5-199
REQUIRE declaration
in behavior models 2-18
REQUIRE statement 3-52
Resistance
specifying in RESISTOR statements 1-11
Resistance-set-names
in the process parameter file F-11
Resistance units
in structural models 3-56
RESISTANCE_SECTION
in the process parameter file F-7
RESISTOR keyword 1-11
RESISTOR statements
See MOS primitives
RESOLUTION statement 3-45
RESTORE command 5-175
RESUME command 5-176
RETURN statement 2-45
REVISION declaration
in behavior models 2-18
REVISION statement 3-54
ROUTINE declaration
in behavior models 2-23
Routines
See Behavior routines
%ROUTINEVALUE 2-46, 5-14
Scheduling (cont'd.)
in behavior models 2-65 2-66
SCOPE command 5-179
SELECTALL statement 2-44
SELECT command 5-182
SELECTONE statement 2-43
SELECT statement 2-42
Semicolons
in behavior macros 2-19
in behavior models 2-5
SET RADIX command
and integer numeric values 4-13
and logic values 4-12
SET SCOPE command 4-23
SET TIMING command
selection of Min, Nom, Max values 3-51
Setup checkers
See also Hazard checkers
defined in a model 1-22
defined in the command language 5-99
violations and boundary conditions 4-48
SHARE command 5-184
Shift operators D-33
SL0 D-33
SL1 D-33
SLR D-33
SR0 D-33
SR1 D-33
SRR D-33
SHOW ACTIVATE command 5-2
SHOW BREAK command 5-12
SHOW CALL command 5-13
SHOW CONFIGURE command 5-23
SHOW CONTEXT command 5-25
SHOW DETECT command 5-52
SHOW DROP command 5-56
SHOW ERROR command 5-61
SHOW FAULT command 5-84
SHOW HAZARD command 5-105
SHOW INDIRECT command 5-117
SHOW KEY command 5-122
SHOW LOG command 5-141
SHOW MACRO Command 5-148
SHOW MOS command 5-151
SHOW PATTERN command 5-153
SHOW RADIX command 5-169
SHOW SCOPE 5-179
SHOW STATE command 5-200
SHOW TERMINAL command 5-208
SHOW TIME command 5-210
SHOW TIMING command 5-212
Index

SHOW TRACE command • 5–228
SHOW WATCH command • 5–246
Shrinkage
specifying in MOS models • 1–12
Signal declarations • 3–26 to 3–45
definition of • 3–25
for wire–tying • 3–31 to 3–37
scope of • 3–26
Signal names • 4–25
implicit assertion/polarity synonyms • 4–26
implicit label,pin • 3–15
in a hierarchy • 4–27
in behavior, network, and command languages • C–2
in instance statements • 3–14
machine descriptors used with • 4–30
preferred • 4–27
search order • 5–180
using quotation marks with • 1–6
Signals
accessing from the command language • 4–20
accessing names and values • 5–207
accessing with wildcards • 4–23
applying pattern vectors to • 5–153
bidirectional • 3–7
comparing actual values to a file • 5–153
displaying state of • 5–66
inputting values to • 5–32
monitoring signal activity • 5–214, 5–238
unnamed • 4–21
SIMBREAK prompt • 5–7
SIMULATE command • 5–192
Simulation
specifying a stop condition for • 5–194
Simulation data
compressing • 5–177
reloading • 5–175
replaying • 5–173
saving • 5–177
Simulation time
resetting and displaying • 5–210
Single quotation marks
in behavior macros • 2–20
in model names • 3–5
in model parameter lists • 3–6
in text strings • 2–1
SLB files • 2–11
S–modules
in the ZYCAD LE • E–1
Soft DECSIM
handling of undefined memory bits • 1–17
Soft DECSIM (cont’d.)
level–sensitive memory in • 1–13
Source files
See Network description files
SPAWN command • 5–196
SPR (a command macro) • 5–199
STATE command • 4–20, 5–200
STATE declaration
in behavior models • 2–14
Statements
See Behavior statements, Structural statements, Instance statements
States
See Behavior states, Command language states
STATIC modifier • 2–35
Statistics
network • 5–202
process • 5–233
runtime • 5–202
simulation • 5–202
STATISTICS command • 5–202
STEP command • 5–205
Strings
See Text strings
Structural models
bidirectional signal declarations • 3–7
compiler lexical statements
INFORM • 3–53
REQUIRE • 3–52
REVISION • 3–54
compiling • 5–18
connection lists • 3–6
pin–based • 3–7
signal–based • 3–6
defining • 3–5
formal parameters • 3–5
global declarations • 3–8
header • 3–5
model body • 3–8 to 3–9
model name • 3–5
names • C–4
nesting of • 3–10
network–level declarations • 3–45 to 3–47
DEFAULT_TIMEOUT • 3–46
MAXRES • 3–47
PRECISION • 3–45
RESOLUTION • 3–45
passing parameters in • 3–47 to 3–52
signal declarations • 3–25 to 3–52
BIDIRECTIONAL • 3–26
CAPACITANCE • 3–38

Index–15
Index

Structural models
signal declarations (cont'd.)
  DELAY • 3-41
  EXPLICIT_NODE • 3-44
  INPUT • 3-29
  IN_STUB • 3-27
  NO_CHARGE_SHARE • 3-43
  NO_WIRETIE_CHECK • 3-36
  OUTPUT • 3-29
  OUT_STUB • 3-27
  PARASITIC • 3-39
  PCAP • 3-40
  POWER • 3-37
  PULLDOWN • 3-36
  PULLUP • 3-36
  SYNONYM • 3-30
  THRESHOLD • 3-40
  TIMEOUT • 3-35, 3-43
  TRISTATE • 3-33
  UNCONNECTED_OK • 3-15, 3-28
  WIREAND • 3-32
  WIREMOS • 3-33
  WIREOR • 3-32
  WIRETIE_MODEL_LEVEL • 3-34
  subblock names • C-4
  text macros • 3-53
  units in • 3-55

Structural statements
See also Instance statements
  INFORM • 3-53
  in subblocks • 3-9
  LIBRARY • 3-2
  PRECISION • 3-45
  REQUIRE • 3-52
  RESOLUTION • 3-45
  REVISION • 3-54
  text macros • 3-53
  units in • 3-55

STRUCTURE/ENDSTRUCTURE keywords • 3-8

Subblocks
  in behavior models
    names • C-4
  in structural models • 3-8 to 3-9
    body • 3-9
    names • 3-8, 3-9, C-4
    specifying in instance statements • 3-11

Subexpressions
  as expression operands • 4-20

Subscripts
  bit subscript operator • D-14
  defaults in behavior states • 2-14

Subscripts (cont’d.)
  for command language states • 4-20
  for fast execution • 2-14
  in behavior models • 2-4
  in behavior routine headers • 2-24
  SXT operator • D-26

Symbols

See also Names
  accessing from the command language • 4-20, 4-22
  accessing with wildcards • 4-23
  in behavior, network, and command languages • C-1 to C-4
  integer values • C-3
  logic state values • C-3
  parameter time values • C-3
  system keywords • C-4
  system variables • C-4
  text strings
    in behavior, network, and command languages • C-3
    types of in DECSIM • 4-20, 4-25 to 4-30

SYMBOLS (a command macro) • 5-207
SYNONYM declaration
  in behavior models • 2-16
SYNONYM signal declaration • 3-30
System keywords • C-4
System variables • 4-15 to 4-20, C-4
  %ARRAY_ELEMENT • 5-223, 5-242
    as expression operands in the command language • 4-15
  %ROUTINEVALUE • 2-46, 5-14
    used with PRINT command • 5-165
ZYCAD-specific • 4-18 to 4-20

T

T
  TIME command • 4-3

Tab prompting
  See Escape prompting

TC directive
  in behavior models • 2-63
TERMINAL command • 5-208
Text concatenation
  in structural models • 3-54
Text macros
  in structural models • 3-53
$TEXT macro template parameter • 5-145
Text strings

Index-16
Text strings (cont'd.)
   as command arguments • 4–9
control characters in • 2–1, 4–9
   in behavior, network, and command languages •
   C–3
in behavior models • 2–3, 2–50
   " or " in • 2–1
special characters in • 4–9
THRESHOLD declaration • 3–40
%TIME
   as a command argument • 4–14
   as print element in %PRINT statement • 2–50
TIME command • 5–210
Time expressions
   as command arguments • 4–13
   in ACTIVATE statement • 2–47
   in behavior models • 2–64
   in delayed assignments • 2–52, 2–62
   in WAIT statements • 2–51
$TIME macro template parameter • 5–145
TIMEOUT declaration • 3–43
TIMEOUT signal declaration • 3–35
TIMESCALE directive in behavior models • 2–60
Time units
   in behavior models • 2–63
   in structural models • 3–55
Time values
   as command arguments • 4–13
Time wheel • 2–65
TIMING command • 5–212
Timing delay
   in behavior models • 2–63
   actual parameters • 2–65
   expressions • 2–64
   formal parameters • 2–64
   literals • 2–63
Timing directive
   in assignment statements • 2–52
Timing verification • 4–39
Toggling
detection of • 5–36
TRACE command • 5–214
Trace formats • 5–221
   qualifiers • 5–222
Trace triggers • 5–222
Transistor ID strings • C–4
Tristate operators • D–35
TRISTATE signal declaration • 3–33
TSCON operator • 3–25, D–35
TS operator • 3–25, D–35
Two's complement arithmetic • 2–63
TWOSTATE modifier • 2–29

TYPE (a command macro) • 5–229

U
ULD files • 5–130
Unary minus
   expression operator • D–8
UNCONNECTED_OK modifier • 2–34
UNCONNECTED_OK signal declaration • 3–28
Undefined memory bits
   handling of • 1–17
Units
   in structural models • 3–55
Unsigned arithmetic • 2–63
UNTIL command • 5–248
UNTIL–DO statement • 2–40
Usage names
   in the process parameter file • F–11
US directive
   in behavior models • 2–63
USE command • 5–231

V
VAX/VMS commands
   executing within DECSIM • 5–29, 5–196, 5–236
VAXSTATISTICS command • 5–233
Vector signals
   concatenating • 3–17
   extracting in an instance statement • 3–19
   in instance statements • 3–17
VMS event flags • 5–184, 5–187
   routines for accessing • 5–189
VMS_COMMAND command • 5–236

W
WAIT statement • 2–51
Watch actions • 5–241
   activating • 5–242
   halting execution of • 5–90
   triggering with assignments • 2–52
WATCH command • 5–238
WHILE command • 5–248
WHILE–DO statement • 2–39
Wide data
Index

Wide data (cont’d.) compared to 32-bit data • 2–14
WIDTH directive • D-4
  in %PRINT statement • 2–49
  in behavior models • 2–60
WIDTH operator • 4–15
Wildcards
  in command language expressions • D-5
  in file names • 4–10
  in machine descriptors • 4–31
  in symbol names • 4–23
  question mark • 4–12
WIREAND signal declaration • 3–32
WIREMOS signal declaration • 3–33
WIREOR signal declaration • 3–32
Wiretie partition rule • E-2
WIRETIE_MODEL_LEVEL signal declaration • 3–34
Wire-tying • 3–21 to 3–25
  in different model types • 3–22
  insertion of phantoms in a network • 3–22
signal declarations • 3–31 to 3–37
  consistency checks for • 3–36
  NO_WIRETIE_CHECK • 3–36
  PULLDOWN • 3–36
  PULLUP • 3–36
  TIMEOUT • 3–35
  TRISTATE • 3–33
  WIREAND • 3–32
  WIREMOS • 3–33
  WIREOR • 3–32
  WIRETIE_MODEL_LEVEL • 3–34

Z

ZBR files • 5–192, 5–250
ZEROFILL directive IN %PRINT statement • 2–49
ZOF files • 5–192
ZXT operator • D–26
ZYCAD • 4–37
  avoiding bottlenecks • E-2
  edge–triggered memory • 1–13, 4–37
  handling of undefined memory bits • 1–17
  initializing memory • 5–127
  LE architecture • E–1

ZYCAD (cont’d.)
  logicals • 4–38
  output files • 4–38
  partitioning commands • E–3
  partitions • E–1
  by the DECSIM/ZYCAD interface • E–2
  creating a PAI file • E–3
  hard • E–2
  tuning networks • E–6
ZYCAD command • 5–250
ZYCAD–style
  edge–triggered memory in • 1–13
  handling of undefined memory bits • 1–17
ZYCAD_PARTITION command • E–3
ZYCAD_S_MAP command • E–4
ZYCAD_S_MODULES command • E–3

X

XOR operator • D–18

Index–18
Reader's Comments

Your comments and suggestions will help us improve the quality of our documentation. Please note that this form is for comments on documentation only.

What I like best about this manual:
____________________________________________________
____________________________________________________
____________________________________________________

What I like least about this manual: ______________________________________________________
____________________________________________________
____________________________________________________

I need more information on: ______________________________________________________
____________________________________________________
____________________________________________________

I found the following errors in this manual:

<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Additional comments or suggestions for improving this manual:
____________________________________________________
____________________________________________________
____________________________________________________

I am using Version ____ of the software this manual describes.

Please indicate the type of user/reader that you most nearly represent:

☐ Electrical Engineer     ☐ Software Engineer
☐ Technical Writer        ☐ Software Support
☐ System Manager          ☐ Other (please specify) ______________

Please indicate the type of work you do:

☐ Custom chip design     ☐ Software development
☐ Semicustom chip design ☐ Support: _____________________
☐ Board or module design ☐ Other (please specify) ______________

What other design tools are you using?
____________________________________________________

Name ___________________________ Date ______________
Mail Stop _______________________ DTN ______________
ENET Address ____________________

1/89 FTM
INTEROFFICE MAIL

SEG/CAD Documentation Group

HLO2-2/G13